ParaLite: a Parallel Database System for Data-intensive Workflows

(Paralite：データ集約的ワークフローのための並列データベースシステム)

Doctoral Dissertation

博士論文

Ting Chen

陳　婷

Submitted to Department of Information and Communication Engineering,
Graduate School of Information Science and Technology,
The University of Tokyo
in partial fulfillment of the requirements for the degree of
Doctor of Information Science and Technology

Thesis Supervisor: Kenjiro Taura(田浦 健次朗)

平成 25 年 6 月 14 日提出

Abstract: Data-intensive workflows have become one of the most important and necessary tools for data-intensive applications since they facilitate the composition of individually developed executables, making it easier for domain experts to focus on their research rather than computation managements. A workflow generally consists of a set of jobs with their dependencies. Since a job is typically an existing executable, data transfers between jobs are generally handled by the workflow system. Usually, data are stored in files and implicitly transferred through a shared file system or explicitly moved by a staging subsystem. Such file-based workflows are often very complex with many jobs due to the low-level description. To schedule a job to computing resources for parallel execution, the input of the job is generally split into multiple small files, thus, leading to a large number of intermediate files.

While there is a critical need for workflow systems to manage scientific applications and data, parallel database systems which have been commercially available for decades and proved to be efficient large-scale data processing platforms, are well-suited to deal with specific aspects of workflow management. Some workflow management systems utilize database technologies to provide functionality such as simplifying the description of a workflow with SQL queries, improving the performance of the execution and facilitating the management of data. While database systems with high-level SQL queries simplify the description of workflows, they generally lack a good support for directly invoking executables from SQL statements. Many of executables are third-party components that receive a large amount of development efforts from the community and are usually developed in a variety of languages. As a workflow is typically built out of such executables, integrating them into SQL statements is very important. Most databases execute the executables in the form of user-defined functions or stored procedures. Thus, programmers who want to invoke such executables as part of SQL statements have to write and compile them with respect to the strict specifications of databases, and are usually constrained by the languages they can use. It is obviously unreasonable for scientists to rewrite their applications with a large number of such executables to allow them to be run by a database. Another limitation of database systems for workflows is inefficient fault tolerance mechanisms. The conventional approach in most existing database systems which handle failures by aborting the query and restarting it from the beginning, is not efficient for long-running jobs in workflows.

To tackle these problems, we propose *ParaLite*, a shared-nothing parallel database system which facilitates the development of workflows and improves the performance of their executions. The basic idea behind ParaLite is to provide a

coordination layer to glue many SQLite instances together, and parallelize an SQL query across them. With ParaLite, jobs in a workflow are expressed with SQL queries and all intermediate data are stored as relational tables. To allow the direct invocation of external executable from SQL statements, ParaLite provides seamless integrations of external executables (*User-Defined Executable*, UDX for short) into SQL statements. The syntax of an UDX is similar to that of a User-Defined Function (UDF) but more flexible in the format of input and output data. With the support of UDX, programmers do not need to write any program with respect to strict specifications of databases. To provide efficient parallel execution of UDXes, ParaLite is equipped with a concept of *collective query*, an SQL query issued by multiple computing clients who collectively receive the results of the query and process them in parallel using UDXes. Collective query enables the co-allocation of computing clients and data sources (data nodes in databases) with consideration of data locality and load balance across all clients. With collective queries, the execution of an UDX is not bound to database nodes and it can be distributed to arbitrary clients for larger scale execution and computational load balancing.

Moreover, for long-running jobs in a workflow, ParaLite supports intra-query fault tolerance with a *selective checkpointing mechanism*, enabling to resume queries from middle of the execution upon a failure. Each query is represented by a DAG of relational operators in which data are typically pipelined between operators. The goal of the mechanism is to find a set of operators whose outputs are worth being checkpointed to minimize the expected completion time of the whole query. It firstly provides a cost model to estimate the expected completion time of a whole query plan under a given failure probability for each operator. Then a divide-and-conquer algorithm is proposed to find a close-to-optimal solution to the problem. The algorithm divides the query plan into sub-plans with smaller search spaces. For a given query plan with n operators, the algorithm runs in $O(n)$ time.

The experimental results firstly show that while ParaLite has similar performance with a commercial database system DBMS-X for most queries from TPC-H benchmark, it is 10x speedup comparing to UDF implementation in DBMS-X for the execution of executables. Besides, ParaLite has several times higher performance than a MapReduce system (specifically Hive) for typical SQL tasks, such as selections, joins and aggregations. With collective queries the performance for the UDX's execution could achieve close-to-ideal speedup with the increase of computing clients when data are either balanced or not balanced distributed across a cluster. Moreover, the mechanism of collective query balances the load across computing clients even when some clients are manually loaded. The experimental results also indicate that different fault-tolerant strategies affect the overall runtimes of queries.

Our selective checkpointing mechanism can choose reasonable operators to be checkpointed and outperforms other fault-tolerant strategies, such as pure pipelining data and checkpointing all intermediate data. In addition, the divide-and-conquer algorithm taken by our mechanism has a smaller overhead than brute-force approach while keeping a similar effectiveness.

Finally, we study three real-world text-processing workflows in the field of Natural Language Processing (NLP), and build them on top of ParaLite, Hadoop, Hive and regular files. We discuss their strengths/weaknesses both in terms of programmability and performance for each workflow. Our development experience reveals that high-level query languages such as SQL of ParaLite and HiveQL of Hive are helpful for expressing data selection, join, aggregation and calculation by typical executables. In NLP workflows, the expressiveness of SQL in ParaLite is particularly useful since it provides natural supports of file-based NLP executables and reusing existing NLP tools by tracking the association between a document and its annotation attached by the tools. On the other hand, workflows expressed in low-level languages lack good support of all features mentioned above, requiring a few extra efforts. The experimental results show that essentially each system has a similar overall performance because the executables dominate the exuecution time. However, a closer investigation still reveals a potential advantage of ParaLite due to data partitioning and query optimization.

Keywords: Data-intensive workflow, MapReduce model, parallel database system, user-defined executable, collective query, intra-query fault tolerance

Acknowledgments

I would like to thank all the people who have helped me for the completion of this thesis.

I want to express my deepest gratitude to my supervisor, Prof. Kenjiro Taura for suggesting me the guideline of this thesis and guiding me during the whole research. I have been extremely lucky to have such a responsible supervisor who cares about my work a lot and discusses with me very frequently. His enthusiasm and excellent ability on programming place a significant influence on my positive attitude to research. In addition, I value the encouragement and confidence that Prof. Taura gave to me every time when I am frustrated by the difficulties during the study. I also appreciate the financial support from Prof. Taura very much.

I am grateful to Prof. Masahiro Goshima and Prof. Yoshihiro Kawahara to be my second advisors who discussed with me on my research once a year and gave me a lot valuable advice on the directions of the research and experiments. I also want to thank Prof. Takashi Chikayama who was concerned about on my life in the laboratory in the first year.

I would like to thank members in Tuara-Laboratory who have helped me in both daily and research lives. Without their help, I could not get used to the life in Japan so quickly. I appreciate their insightful suggestions and discussions on my work during the seminars. In particular, I would like to express my sincere gratitude to Mr. Kentaro Hara and Miss. Miki Horiuchi for their accompanies in my most difficult early days in Japan, Mr. Jun Nakashima and Mr. Shigeki Akiyama for helping me with preparing compulsory activities of doctoral courses, and Mr. Sho Nakatani who offered the most help on my research and taught me a lot of technologies patiently.

I would like to give my thanks to Prof. Masaru Kitsuregawa, Prof. Jun Adachi, Prof. Masashi Toyoda, Prof. Yoshimasa Tsuruoka and Prof. Masahiro Goshima for reviewing my thesis. Specially I want to thank Prof. Masaru Kitsuregawa who is an expert in the database community for giving me many valuable suggestions for my researches and encouraging me in the entrance examination of the university, Prof. Masashi Toyoda and Prof. Yoshimasa Tsuruoka for introducing me some related papers.

Finally, I must express my heartily gratitude to my parents and my husband, Cheng Luo, for their continued support and encouragement during my whole life. My husband taught me about how to keep a positive and optimistic attitude when I faced difficulties which I benefited from a lot.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Motivation

The fourth paradigm of science, data-intensive computing [51], is characterized by the big exploding of data. Researches in many disciplines, such as environmental science, astronomy, particle physics, and medicine, are increasingly relying on data-intensive computation. There is a wide agreement that data-intensive methods are key to these applications and receive more and more interest [7]. Such methods are expected to play more and more important role in providing evidence for well-informed policies and decisions. Therefore they are of great scientific and social importance. There are many approaches to support data-intensive methods, such as workflow systems [35, 5, 104] and MapReduce framework [32] over large-scale distributed resources.

Among all these methods, data-intensive workflows have become one of the most important and necessary tools for data-intensive applications since they facilitate the composition of individual executable, making it easier for domain experts to focus on their researches rather than computation management. Workflows are widely used to process a large number of text, especially in the discipline of Natural Language Processing (NLP). Common tasks in NLP are to extract the features of data (aspects of the representations of the data) some of which may be superficial, such as the words and sequences of words themselves while others are more complex, such as both the grammatical and semantic relationship between words. To accomplish these tasks with workflows, easy description and parallel processing of tasks readily accessible to NLP scientists. Many systems are proposed to execute workflows, including GXP Make [103], Swift [122], Pegasus [34] and Taverna [77].

A workflow is generally a DAG with a set of independently developed jobs and their dependencies. Each job is a typical existing binary or executable and communicates with another job in the workflow. A data transfer among jobs is generally handled by the workflow system. Usually, data are stored in files and implicitly transferred through a shared file system or explicitly moved by a staging subsystem. A file-based workflow is always very complex with many jobs due to its low-level description. Besides, to process file in parallel, a big file is split into small files, thus, leading to a large number of intermediate files. Users usually tend to complain that it is troublesome for them to manage thousands of files, and also inconvenient to extract useful information from so many files. In addition, using files to store data may lead to poor performance for the execution of a workflow. Since creating index for data stored in files is usually difficult, it is very tedious and inefficient to select a subset of data which requires a full scan to files.

With a common goal of making large scale data processing simple and easy, MapReduce model [32] has attracted wide interests from both industry and academia due to its simple programming model and good scalability across hundreds of nodes. After the emergence of MapReduce and its open-source incarnation Hadoop [116, 53] in particular, more and more efforts are made to enable or utilize them in scientific workflows. Researchers either create workflows with MapReduce features or integrate Hadoop into workflows to get better performance for the executable of jobs. However, MapReduce in general requires users to develop two functions map and reduce; Hadoop requires them to be written in Java conforming the class library framework, at least by default. This low-level description increases difficulties for users to develop their applications. Hence, some MapReduce systems are extended to support high-level language (e.g. SQL-like queries) [106, 44]. In addition, as data are stored in the distributed file system (HDFS in Hadoop), it is difficult to index data too.

While there is a critical need for workflow systems to manage scientific applications and data, and database technologies are well-suited to deal with some specific aspects of workflow management, a nature idea is to build workflows on top of the parallel database system [36]. Parallel database systems have been dominating platforms for large-scale data processing and will continue to be prominent in future. They have been commercially available for nearly two decades and there are now about a dozen in the marketplace, including Teradata [105], Microsoft SQL Server [92], Vertica [111], DB2 [29], and Oracle [81]. They are robust, high performance computing platforms to provide a high-level programming environment and parallelize data processing easily. The DBMS community has also been working on systems customized for data-intensive science applications and has built many

prototypes such as extensions to MonetDB [63], SciDB [96] and Sloan Digital Sky Survey (SDSS [99]) which are successfully being used today. Firstly, with expressive SQL, database systems can simplify the description of workflows. For instance, SQL queries with a proper support of user-defined functions and reductions can express many data processing tasks much more elegantly and easily than MapReduce. Secondly, database systems facilitate the management of data naturally. Finally, databases are efficient for processing relational data in ways expressible in SQL due to data indexing and sophisticated query optimization [82, 97].

While parallel database systems facilitate the description of workflows in terms of expressing jobs with SQL queries and provide efficient management of data, they generally have some limitations:

Non-straightforward Integration of Executable: As a workflow is typical built out of various individually developed executables, integrating such executables into SQL statements is very crucial. For example, NLP workflows typically consist of data scrapers, sentence splitters, part-of-speech taggers, named entity recognizers, parsers, data indexers, and so on. Many of them (e.g., parsers [39, 12]) are third-party components that received a large amount of development efforts from the domain community and are usually developed in a variety of languages. Most databases execute external modules in the form of user-defined functions or stored procedures. Thus, programmers who want to invoke such executables as part of SQL statements have to write and compile them with respect to the strict specifications of databases, and are usually constrained by the languages they can use (e.g. C/C++/Java). It is obviously unreasonable for scientists to rewrite their applications with a large number of such executables to allow them to be run by a database.

Limited Performance in Executable Execution: Another general limitation of parallel database systems is that they do not optimize data transfers between data nodes and parallel clients that process large query results. A significant work exists in minimizing IO costs and data transfers inside the execution of an SQL query [36], but query results are all returned to a single client who issued the query. When big results are returned to a single client and then distributed to external programs for parallel execution, the single client can easily become a bottleneck. Moreover, it prohibits us to take advantage of co-allocating computing clients with data.

Ineffective Fault Tolerance for Long-running Executable: As data that need to be analyzed continues to grow, the size of computing resources grows accordingly. Thus, the probability of a failure during query processing increases rapidly. Most existing database systems handle failures by aborting unfinished queries upon a failure and restarting the entire query processing. This approach is reasonable for queries with OLTP workload as almost all transactions must be completed within a

small amount of time. However, this conventional approach is not efficient for the long running jobs of OLAP workload in workflows because it is time-consuming to restart the query from the beginning as lots of work are lost.

1.2 Contribution

With consideration of the advantages of databases, our goal is to develop workflows on top of a database system *ParaLite*, with which jobs are expressed in SQL queries and all intermediate data are stored as relational tables. ParaLite is a shared-nothing parallel database system which provides an coordination layer to connect multiple single-node databases and parallelizes queries across them. To support workflows better, it has the following distinguished features:

- *Straightforward integration of executable.* ParaLite provides *User-defined Executable (UDX)* to make it straightforward to integrate arbitrary executables in a query. UDX considerably lowers users' efforts to describe jobs in workflows as it allows the user to define the executable with the input/output format directly in a query without writing any program. The implementation of UDX also reduce the start-up overhead which would be too heavy for typical NLP programs by invoking the executable on a block of tuples rather than every single one.

- *Efficient parallelization of executable.* ParaLite proposes a concept of *collective query*, a single SQL query issued by many clients who collectively receive the results of the query and then perform arbitrary external executable on them in parallel. With collective query, executables are not bound to database nodes and they can be distributed to arbitrary clients, for larger scale execution and computational load balancing. The concept of collective query optimizes data transfer between parallel compute clients and data sources and makes it easy to tune the parallelism by adding/reducing the number of clients.

- *Intra-query fault tolerance.* ParaLite presents a *selective checkpointing mechanism* which looks for a set of operators whose outputs are worth being checkpointed to minimize the expected runtime of the whole query, enabling to resume queries from middle of the execution upon failures. The mechanism is more effective than uniform strategy of checkpointing nothing (used by most commercial databases) and materializing all intermediate data (used in MapReduce framework).

In addition, we study several real-world text processing workflows and develop them on top of ParaLite, Hadoop, Hive and general files. We discuss their strength-

s/weaknesses both in terms of productivity and performance for each workflow. Our experiences and experimental results reveal some interesting trade-offs: (1) High-level query languages (SQL of ParaLite and HiveQL of Hive) are helpful for expressing data selection, aggregation and calculation by typical executables; (2) To reuse existing NLP tools, it is often important to be able to track the association between a document and its annotation attached by the tool, for which the expressiveness of SQL is particularly useful; (3) Each system has similar performance in the execution of overall workflows because essentially performing executables takes most of the time, but small differences could reveal some potential trade-offs that each system entails for workflows.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows:

Chapter 2 provides a comprehensive understanding of the background of our work, including parallel database systems, MapReduce frameworks, target workflows and the problems of existing parallel database systems. We present related works on how current database systems and MapReduce systems solve these problems in Chapter 3.

In Chapter 4, we present our major work ParaLite, a shared-nothing parallel database system. The details of the integration of executables into SQL statements and the parallelization of them are introduced in Chapter 5 while the intra-query fault tolerance mechanism is elaborated in Chapter 6.

Chapter 7 gives the evaluation of ParaLite to verify the scalability and performance of the system and the effectiveness of proposed mechanisms.

In Chapter 8, we show our efforts on three real-world text-processing workflows built on top of ParaLite, Hadoop, Hive and general files and compare them in terms of both productivity and performance.

Finally, we give our concluding remarks and suggest future directions of our research in Chapter 9.

Chapter 2

Background

Contents

2.1 Parallel Database Systems and MapReduce Frameworks

2.1.1 Architecture of Parallel Database System

A parallel database system provides the same functionality as centralized databases except in an environment where data are distributed across the nodes of a cluster or processors of a multiprocessor system [36, 124]. Ideally, a parallel DBMS should demonstrate two advantages: linear scaleup and linear speedup. Linear scaleup refers to a sustained performance when both database size and processing capacities

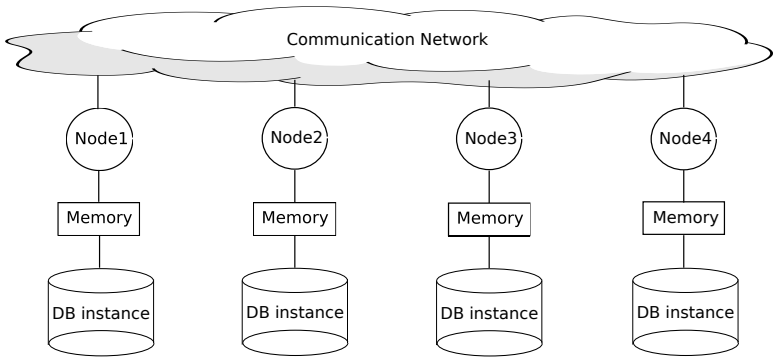

Figure 2.1: Architecture of a Shared-nothing System

are linearly increased. Linear speedup refers to a linear increase in performance when the database size keeps unchanged but the processing capacities are linearly increased.

Most parallel database systems are based on a cluster of commodity computers called “ shared-nothing nodes ” as Fig. 2.1 shows. In the shared-nothing architecture, each node has its own memory and one or more disks. Nodes communicate with each other through a high-speed interconnect network. Data storage in such architecture is distributed among the nodes by connecting them. The shared-nothing architecture reduces interference by minimizing resource sharing and data transfer. Memory and disk accesses are performed locally on each node, and only the filtered (reduced) data is passed to the client. Since shared-nothing architecture provides high scalability due to involving minimal interference between nodes/processors and minimal traffic on interconnection network, it is widely used in Parallel DBMSs.

Every parallel database system is built and pioneered by these two techniques: data partitioning and the partitioned execution of queries. The idea behind data partitioning is to distribute the tuples of a relational table across the nodes for parallel execution. There are three typical partitioning methods for data: Round-robin fashion, Range fashion and Hash fashion [36]. With the data partitioned across the nodes of the cluster, a query can be trivially executed in parallel. In Parallel DBMSs, each SQL query is represented as a DAG of operators such as SELECT, SORT, JOIN and so on. These operators take relations as input and produce relations as outputs. The uniformity of the operators and data allow them to be arbitrarily composed into the execution flow. By streaming the output of one operator into another operator, the two operators can work in series giving pipelined parallelism. By partitioning the input data among multiple nodes/processors, an operator can often be split into many independent operators and each works on a part of the data. This partitioned data and execution is called partitioned parallelism.

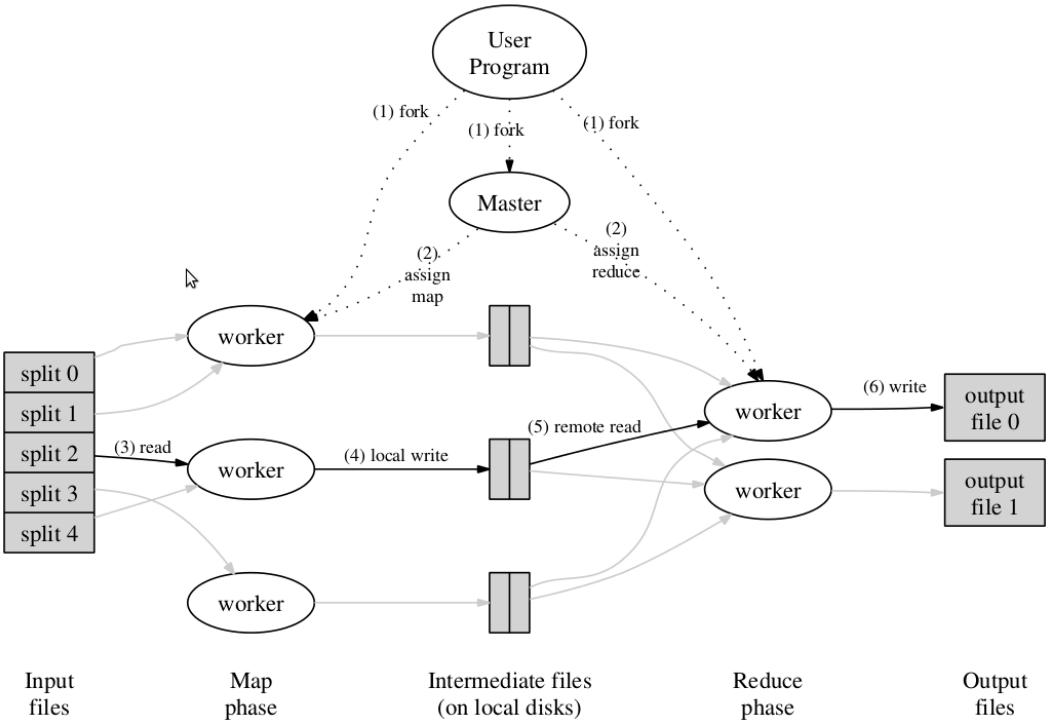

Figure 2.2: The Overview of Execution in MapReduce [32]

2.1.2 Overview of MapReduce

MapReduce [32] is a programming model for processing huge data sets on certain kinds of distributable problems using a large number of computers (nodes). The computation takes a set of input key/value pairs, and produces a set of output key/value pairs which can be presented by two functions: Map and Reduce.

A Map function, written by user, takes a series of key/value pairs, processes each, and generates zero or more intermediate key/value pairs. The input and output types of the map can be (and often are) different from each other. The MapReduce integrates associated values with the same intermediate key and passes them to the Reduce function. A Reduce function, also written by user, reads an intermediate key and a set of values for that key, iterates through the values that are associated with that key and outputs zero or more values.

A task is executed as Fig. 2.2 shows. First of all, the input file is split into M pieces (the size of each piece is controlled by user), starts up many copies of programs on multi machines of cluster. One of the copies of program is master, a special one who is responsible for assigning work to other copies called workers. In generally, there are M map tasks and R reduce tasks. The master picks idle workers and assigns each one a map task or a reduce task. A worker who is assigned

map task firstly reads the corresponding input split. Then it parses key/value pairs out of the input data, passes them to the Map function. After desired filtering and/or transformations, a set of intermediate key/value pairs is produced. Then the Map worker partitions the intermediate key/value pairs into R regions (files) by partitioning function. Hash function is very popular for data partitioning. Since M workers participate in the map phase, there are R files on disk storage at each of M nodes, for a total of $M \times R$ files. The locations of the partitioned pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

When a reduce worker is notified by the master about the intermediate pairs' locations, it remotely reads the data from the local disks of the map workers. The input for each reduce instance consists of the M files. When a reduce worker has read all intermediate data, it sorts the data by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is necessary since typically many different keys map to the same reduce task. If the amount of intermediate data are too large to fit in memory, an external sort is used. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.

When all map tasks and reduce tasks have been completed, the output of the MapReduce execution is available in the R output files (one per reduce task, with file names as special led by the user). All these output files are passed as input to another MapReduce call or another distributed application that is able to deal with input that is partitioned into multiple files typically.

2.1.3 Programming Model

During the 1970s, the database research community engaged in a contentious debate between the relational advocates and the Codasyl advocates [31]. In the end, DBMS uses relational representation (SQL queries) which is stating what you want, rather than presenting an algorithm for how to get it. Programs in high-level languages, such as SQL, are easier to write, easier to modify, and easier for a new person to understand.

For example, assume a table named *document*, with each row an *id* with a *word*. The word count task which is to calculate the occurrences of words can be simply expressed by an aggregation operation on the column *word* as shown in the the following SQL statement:

SELECT word, count(*) as wordcount

FROM document
WHERE word> ' c ' and word < ' t '
GROUP BY word

Many relational databases support User-Defined Functions (UDFs) in which a developer can implement tasks using a procedural language. User-defined functions are longstanding database features that enable database extensibility. UDFs allow customization of how a database processes data, eliminating the limitation of operations brought by a SQL query. For example, an user-defined function *parse* is used in the SELECT statements and invoked on each row to convert the value of the column *word* to a value produced by the UDF.

SELECT id, parse(word) as parse_result
FROM document

On the other hand, MapReduce model presents an algorithm for data access which can be consider as Codasyl. The user is forced to write algorithms for the Map and Reduce function in a low-level language in order to perform record-level manipulation. For example, the word count task requires a Map and Reduce functions which are described in Fig. 2.3.

```
void map (String name, String document):
    //name: document name
    //document: document content
    for each word w in document:
        if w > "c" and w < "t"
            EmitIntermediate(w, 1);
```

```
void reduce(String word, Interator counts):
    //word: a word
    //counts: a list of word counts
    int result = 0 ;
    for each pc in counts:
        result += pc ;
    Emit(AsString(result)) ;
```

Figure 2.3: Word Count Task in MapReduce

2.1.4 Data Storage

Parallel DBMSs require data to fit into a schema such as a relational paradigm of rows and columns. Parallel DBMS advocates think schema is important since the run-time system of the DBMS can ensure that input records obey this schema. This is the best way to keep an application from adding "garbage"to a data set. They also insist that separation of the schema from the application is good. If a programmer wants to write a new application against a data set, he or she must discover the record structure. In modern DBMSs, the schema is stored in a collection of system catalogs and can be queried (in SQL) by any user to uncover such structure.

In contrast, the MapRedcue model does not require that data files adhere to a schema defined by the relational data model. That is, the MapReduce programmer is free to structure their data in any manner or even can have no structure at all. But there exist some potential problems for free data structure when data are shared by multi-applications. Since a programmer has to write functions explicitly to handle raw data, it is difficult for other programmers to know the structure of data unless they analyze the source code of the first programmer. Even if a schema is separated from applications and available to all the programmers, they have to ensure that any updates (addition or modification) on data does not violate some high-level constraints (e.g., the value for a specific column must not be NULL). All these additional works are tedious and troublesome.

After the data are stored, all modern DBMSs create indexes (e.g. hash or B-tree indexes) for data to accelerate access to them. If a user wants to look for a subset of records, the scope of the search dramatically reduces with a proper index. Most database systems also support an index on multiple columns or multiple indexes per table. However, MapReduce frameworks do not provide built-in indexes. Programmers have to implement any indexes that they want to accelerate access to the data for their applications. Creating indexes inside their applications is difficult because the mechanism for reading data from the file system must be changed to use these indexes. Moreover, if there exists index sharing between different programmers, this is an un-acceptable strategy as the similar reason for “no schema data is bad”. The specifications for the creation and usage of indexes must be transferred among programmers. Therefore, it is better to use a standard format to store indexes in the system catalogs. As a result, programmers can discover the information of indexes by simply querying to the system.

2.1.5 Data Flow

The processing of a task with both approaches of parallel database systems and MapReduce systems can be presented by data flows. In parallel database systems, a query is transformed into a DAG of relational operators, such as sort, join, aggregation and selection. Data are simply pipelined between operators. When one operator must send data to the next operator, regardless of whether that the operator is running on the same node or a different one, the qualifying data are “pushed” by the first operator to the second operator. Hence, data are streamed from producer to consumer without being written to disks. Such data transfer pattern is efficient because it doesn't suffer from the bottleneck of disk accesses. However, it is not effective to recover a query from failures, which we introduce in the Section 2.1.7.

On the other hand, in the MapReduce framework, data are moved from Map to Reduce and the intermediate data are stored in disk. The master notifies reducers about the information of the intermediate pairs and then the reducer remotely reads the data from the local disks of the map tasks. This mode is characterized by the patten of "PULL". Consequently, disk size becomes a potential bottleneck. There also exists another concurrency problem. M Map instances can produce M*R intermediate local files. Each of the R Reduce instances needs to read its M input files. With so many of Reduce instances running at the same time, two or more Reduce instances probably attempt to simultaneously read their input files from the same map node, leading to a lot of disk seeks and low disk transfer rate.

2.1.6 Processing Optimization

In Parallel DBMSs, query processing refers to the automatic translation of a query, query optimization and its parallel execution. Query translation takes an SQL query and translates it into a relational algebra. In the process, the query is analyzed semantically so that incorrect queries are detected and rejected as easily as possible, and correct queries are simplified. The query optimization is to find the best execution plan by permuting the ordering of operations. The query optimizer is usually seen as three components: a search space, a cost model, and a search strategy [109] as Fig. 2.4 shows. The search space is the set of alternative execution plans to represent the input query. These plans are equivalent, in the sense that they yield the same result but they differ on the execution order of operations and the way these operations are implemented. The cost model predicts the cost of a given execution plan. To be accurate, the cost model must have accurate knowledge about the parallel execution environment. The search strategy explores the search space and selects the best plan. It defines which plans are examined and in which order. Cost model based optimization mechanism easily optimizes the communication between operators [68, 18]. One example is pushing the WHERE clause down of the execution plan so that data filtering is done before they are joined or aggregated, resulting much less data transfer through network. Another example is deciding the order of join. This optimizer takes the advantage of data partitioning method. It requires each operand relation to be partitioned the same way. For example, if R and S are both partitioned across multiple nodes using the same hash function on the join attributes, the join operation on R and S can be executed locally with much less data transfer since records with the same hashed attribute are in the same nodes. In addition, the optimizer also can get the optimal order for multiple joins based on the estimated number of result tuples for each join.

While parallel database systems provide efficient general cost-model based opti-

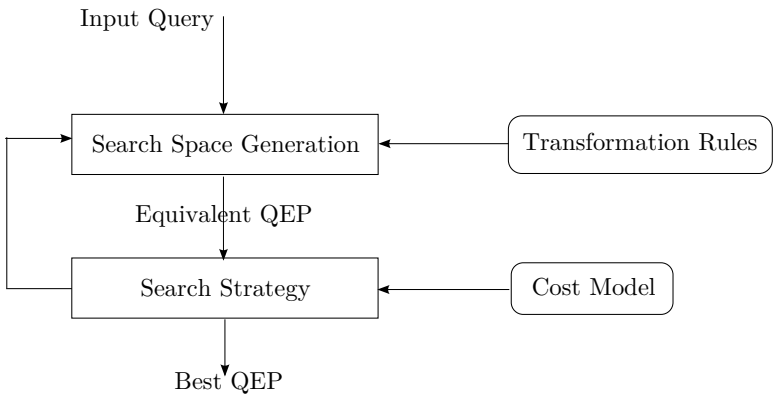

Figure 2.4: Optimization Method in Parallel DBMS

mization to get the optimal solution, MapReduce can support only domain specific optimization for the job execution. A significant bottleneck existing in the execution is the repetition in the intermediate keys from each map task. For example, in the word count task, each map task produces thousanands of records for a single word in form of ⟨word, 1⟩. All of these pairs are sent to a single reduce task through the network and then aggregated together to produce a final result number by the Reduce function. To solve this problem, a common method is to allow users to customize an optional *Combine* function which does partial aggregation of the output data from map tasks. A combiner runs between the mapper and reducer and executes the function on the each node that performs a map task. The output from each mapper is sent to the combiner, which performs the aggregation of the word count for the same word locally in the example above. The output from the combine is then written to disk, before being sent to the reducer. By this way, a lot of data are reduced to be transferred to reducers.

2.1.7 Fault Tolerance

Most existing parallel database systems support transaction-level fault tolerance by aborting unfinished queries upon a failure and restart the entire query processing. Data are usually replicated on multiple nodes through specific mechanisms such as disk shadowing, interleaved declustering and chained declustering [40, 88, 100]. If a single node fails during a running query in a DBMS, the entire query must be completely restarted on the replica node. Part of the reason for this approach is that DBMSs avoid saving intermediate results to disk whenever possible. This approach is reasonable for queries with OLTP workload as almost all transactions must be completed within a small amount of time. However, for long running queries of OLAP workload, it is costly to restart the query from the beginning as lots of work

are lost. This problem should be readily solved by providing operator-level fault tolerance. The runtime system marks one or more operators in the query plan as "checkpointed operators" whose results are saved to disk. Once a failure occurs, the query can be recovered from these checkpointed operators.

On the other hand, MapReduce frameworks provide a more sophisticated failure model than parallel database systems. While both types of systems use data replications to deal with node failures, MapReduce is far more effective at dealing with node failures during the execution. In a MapReduce system, if a Map/Reduce worker fails, the master can automatically reschedule the task that was working on the failed worker to an alternate node to be re-executed. If the master fails, a new copy can be started from the last checkpointed state. The main reason behind this effective recovery mechanism is contributed to the fact that the output of the Map tasks are materialized locally instead of being pipelined to the Reduce tasks. While this strategy is safe-first, it is not always efficient for smaller to medium-sized jobs as writing all intermediate data to durable storage before making progress may not gain from re-execution of tasks.

2.1.8 Development and Setup

The first thing that user care about is how to get a parallel DBMS and MapReduce system. Most of MapReduce systems are open source projects available for free, such as Hadoop. DBMSs, and in particular parallel DBMSs, are expensive; though there are good single-node open source solutions. To the best of our knowledge, there are no robust, community-supported parallel DBMSs.

Even in case that a user has already got some parallel DBMSs, he would be possibly disappointed by the fact that these systems are difficult to install and configure properly. This is because the user often face the difficulty in tuning parameters which must be set correctly for the system to operate effectively. On the other hand, an open-source MapReduce implementation provides the best “out-of-the-box” experience. Thus, it is faster for users to get the MapReduce system up and run queries than the DBMSs. Once a DBMS is started and running properly, programmers have to write a schema for their data (if no schema exists for the data) and then load the data into the system. The time spent on this process is considerably larger in a DBMS than in a MapReduce system. While the DBMS has to parse and verify each datum in the records, MapReduce programmers load their data to the system just simply by coping it into the underlying distributed storage system.

However, although it may be easier to for users to get started with MapReduce, it may bring significant pain to applications developers for maintenance of MapReduce programs. Data increases a lot with the development of application.

In a MapReduce system, it is necessary to modify the existing MapReduce code and retest them to ensure that the new MapReduce programs work with the new assumptions about the data's schema. In contrast, once DBMS users have built the initial SQL-based applications, they do not need to modify the code despite several changes to new schema.

2.2 Data-intensive Workflows

A scientific workflow is typical formed by connecting multiple jobs based on their dependencies. A workflow is typical represented as a Directed Acyclic Graph (DAG) which indicates the relationships between jobs as Fig. 2.5 shows. In a workflow, one job starts after a previous job has completed and jobs without any dependency can be performed concurrently. Jobs are executables written in various languages in most workflow systems and web services in a few workflows. In this section, we introduce four key issues in a workflow management system – workflow composition, workflow scheduling, data movement and fault tolerance.

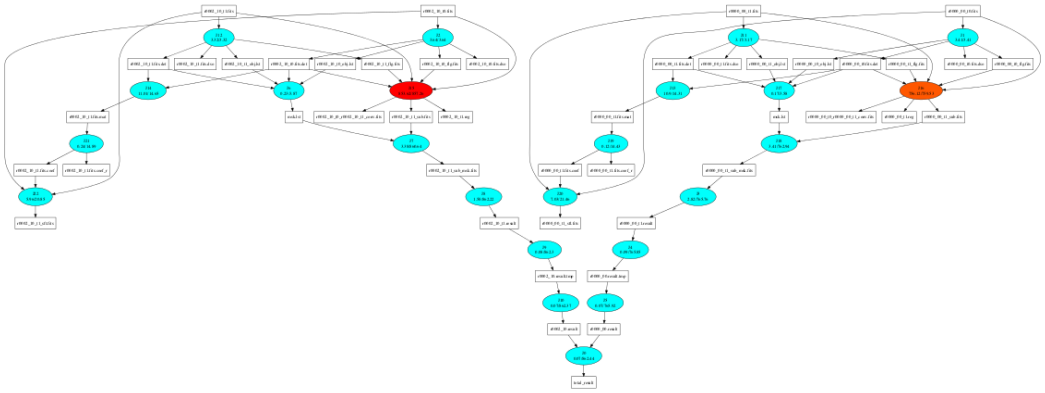

Figure 2.5: An Example of Worklfow [91]

2.2.1 Workflow Composition

The composition of workflows is the first and important step of the entire workflow process. It allows a user to specify the jobs and their dependencies. Usually there are two types of fashions, namely *abstract* and *concrete*. With the abstract fashion, a workflow is specified without considering the underlying resources for its execution while with the concrete fashion, workflow jobs are bound to specific resources.

Many workflow systems use a particular language or representation to describe a workflow, such as BPEL [67], Swift [122], DAGMan [26] and GXP make [103]. The

languages they provide can be composed manually using a plain text editor with a specification. While language-based composition works well in some systems, writing the textual programs by hand is very difficult. The situation becomes worse if the compositional language cannot support standard programming control components well. To solve this problem, a user can use a high-level script (e.g. Python and Ruby) to express complex control and generate the lower-level primitives of workflows.

To simplify the composition for scientific users, some workflow systems provide graphical tools for composing workflows, such as Taverna [77], Triana [104] and Kepler [5] who build workflows with composing graphs where the nodes represent jobs and edges represent jobs' dependencies. Compared to language-based composition, graphical composition is more intuitive and can be handled more easily even by a general user who is not an expert or sophisticated programmer. While graphical composition is easy to describe workflows with small number of jobs, it is general troublesome for describing more complex workflows with a few dozen jobs. As a result, most graphical tools support graphical nesting forms based on sub-workflow hierarchies. In addition, graphical composition has complexity in expressing concurrency in a workflow (e.g. "for-each" function). Some systems provide specialized control primitives to address this problem.

2.2.2 Workflow Scheduling

Workflow scheduling focuses on mapping a workflow onto distributed resources and managing the execution of workflow jobs. Usually there is a central scheduler who makes scheduling decisions for all jobs in the workflow. While such centralized scheduling can produce efficient policy since it has all information of all jobs, it is not scalable with respect to the number of resources and jobs.

Firstly, as workflows have different features, it is difficult to find a standard best mechanism for mapping workflows onto resources for all workflows. There are two kinds of decisions about mapping jobs onto resources: local decision which is made based on the information of current job, and global decision which is made based on the information of the entire workflow. While scheduling based on global decisions provides better overall performance, it takes much more time to make the decisions. The overhead from the global decision making decreases the overall performance. Thus, both the overall execution time and overhead for decision making should be considered in the scheduling. In general, mapping a workflow onto distributed resources is a NP-complete problem. Some systems use a heuristic solution to obtain near-optimal solutions to satisfy the QoS constraints, e.g. deadline and budget.

After making the decisions for mapping a workflow onto resources, the next step is workflow planning which translates the abstract model to concrete model referring

to the resources. The translation is conducted in either static or dynamic fashion. The static fashion does not take the dynamically changing status of resources into consideration and the concrete model is generated before the execution of workflow based on the current information of the resources. The static fashion is based on user-directed or simulation-based scheduling. In the former, users simulate the process and map the workflow onto resources based on their knowledge, requirement or specific performance criteria. In the simulation-based scheduling, the resource mapping is decided by simulating the workflow execution on a set of resources before it is really executed. In contrast, in a dynamic fashion, both static and dynamic information of resources are taken into account and the concrete model is generated at run-time. Some dynamic fashions use prediction-based scheduling which makes the mapping decisions based on dynamic information and predicted results. Others use in-time scheduling which makes a scheduling decision when a task is executed rather than making the decision prior to the scheduling.

2.2.3 Data Movement

For a workflow running on distributed resources, the output files of jobs may be required by its successor jobs which are processed on remote resources. Thus, the intermediate data have to be staged out to the corresponding remote resources. Most workflow systems support intermediate data transfer automatically through a shared file system or a file staging system [66] while others require users to manage the transfer conforming to the specification of the workflow.

The approaches of automatic intermediate data transfer can be classified as centralized, mediated and peer-to-peer [119]. In the centralized method, the intermediate data are transferred through a central point. While the centralized approach is easily implemented, it is not efficient in data transfer and only suitable for the workflows in which only small-scale data are required to be transferred. A mediated approach uses a distributed data management system to manage the locations of the intermediate data. Mediated approach is scalable and suits workflows with the requirement of large-scale data transfer. A peer-to-peer approach transfers the intermediate data between resources. Since data are transmitted between resources directly, peer-to-peer approach reduces the transmission time significantly. It also releases the bottleneck caused by the centralized and mediated approaches due to not involving any third-party software. Therefore, the peer-to-peer approach suits large intermediate data transfer better.

2.2.4 Fault Tolerance

In distributed environments, various failures can occur during the execution of a workflow such as network failure, disk failure and process failure. Thus, workflow systems should be able to detect and be tolerant of failures by supporting efficient recovery from those failures. To achieve fault tolerance, the combination of checkpointing and rollback recovery mechanism are exploited. Checkpointing approach periodically stores the state of the system during its execution as checkpoints or snapshots, whereas rollback recovery, once a failure occurs, provides a way of restarting a system from the previously saved state. Scientific workflow systems incorporate the checkpointing and rollback recovery schemes at different granularity levels: task-level and workflow-level [84, 60].

Task-level fault tolerance, widely used in parallel and distributed systems, saves the intermediate states of running tasks, so that a task can be re-started from a previously saved state in case of a failure. It restarts the failed tasks transparently on other resources upon resource failures, so that the task can continue its execution from the point of failure. Besides checkpoint/recovery technology, task-level fault tolerance has another simpler policy called retry. The retry technique, the simplest recovery technique, simply tries to execute the same task on the same resource or another resource after a failure occurs. Workflow-level fault tolerance captures the state of the workflow as a whole. Once a failure occurs, the execution of the whole workflow restarts from the last saved checkpoint. In addition, the system can achieve fault tolerance by the technique of redundancy which executes multiple alternative tasks at the same time.

2.3 Why a Database is Essential for Workflows?

From the user's point of view, there are several important aspects existing in workflows as follow:

- How to describe the workflow and data?
- How to invoke the workflow?
- How to monitor the workflow?

To make it easy for a user to fulfill the above requirements, the following activities should be handled well. Meanwhile, database technologies are well-suited to deal with them.

- Workflow Description: As described in Section 2.2.1, many workflow systems use a particular language or representation (usually in a low-level way) to

describe a workflow. Such low-level description brings difficulties for users when they want to develop and reuse jobs in the workflow. Database systems support high-level declarative SQL queries. Programs in high-level languages are easier to write, modify, and understand for a new person. Describing a workflow with SQL queries can simplify its complexity by reducing the jobs in the workflow. Usually, several MapReduce jobs can be easily expressed with a single SQL query.

- Workflow Optimization: To maximize the efficiency and throughput of the system, some optimizations on the job execution and data transfer are required. Meeting such requirements becomes more and more important as the scale of a cluster increases. Databases have been solving the problem of optimizing queries over distributed resources to minimize the overhead of network and disk I/O and cpu time. In addition, a lot of works on the dynamic query optimization are proposed. They adjust the query plan based on the feedback from the execution of queries. Therefore, database technologies are well-suited to optimize the execution of scientific workflows in distributed and dynamic environments.

- Data Management: A workflow system must keep track of the origins of data and their movements during the execution. In addition, it has to provide the functionality of managing the replicas and consistency of data, and also has to provide data recovery upon a failure. Databases can meet all these requirements easily. The support of transactions keeps data consistency when concurrent operations happen on data. Transaction-level fault tolerance also keeps the consistency of data when a failure occurs.

- Concurrency Control: With the increase of the size of cluster and workload, the interference between jobs being executing simultaneously increases. Thus it is necessary to recognize and prevent such interference which has not been solved adequately in workflow systems. Databases naturally deal with the concurrent accesses at different level of granularity and data consistency. They also provide different degrees of transaction consistencies.

- Query Capabilities: In addition to data from users, a large amounts of operational or cluster data are generated, including the status of machines and jobs in the cluster, the information for file access and the usage of resources. Such data are difficult to administer if they are stored in files. The tasks of managing a set of underlying resources, monitoring their status and diagnosing their health could be simplified if all these catalog data are stored in databases and

accessed by SQL queries. Databases also provide the security on data belong to different users. A user is allowed to query on their personal data without interfering the privacy of others.

2.4 Limitations in a Database for Workflows

- Non-straightforward Expression of Executables: One advantage of database systems for workflows is the expressive high-level SQL query which can simplify the description of workflows. While SQL queries can easily express tasks such as selection, aggregation and join on relational data, they are generally difficult to express various individually developed executables. Many of such executables are third-party components that have received a considerable amount of development efforts from the community and usually developed in various languages. As a workflow is typically built out of such executables, integrating them into SQL statements is very important. Most databases execute external modules in the form of user-defined functions or stored procedures. Thus, programmers who want to invoke such executables as part of SQL statements have to write and compile them conforming to the strict specifications of databases, and are usually constrained in the language they can use (e.g. C/C++/Java). It is obviously unreasonable for scientists to rewrite their applications with a large number of such executables just to allow them to be run by a database.

- Limited Performance in Executable Execution: Another general limitation of parallel database systems is that they do not optimize data transfers between data nodes and parallel clients that process large query results. A significant work exists on minimizing IO costs and data transfers inside the execution of an SQL query, but query results are all returned to a single client who issued the query. When big results are returned to a single client and then distributed to external programs for parallel execution, the single client can easily become a bottleneck. Moreover, it prohibits us to take advantage of co-allocating computing clients with data. In addition, as typical NLP programs have high start-up overhead, invoking the executable on every single tuple for most existing database systems would heavily degrade the overall performance.

- Ineffective Recovery for Long-running Jobs: As each job is expressed by a SQL query and processed by the database system in parallel, failures that occur during the execution of the query are handled by the database system. Most existing database systems achieve fault tolerance by aborting unfinished

queries upon a failure and restarting the entire query processing. This approach is reasonable for queries with OLTP workload as almost all transactions must be completed within a small amount of time. However, this conventional approach is not efficient for long running jobs in workflows because it is costly to restart the query from the beginning as lots of work are lost.

Chapter 3

Related Work

Contents

3.1 Large-scale Data Processing

3.1.1 Parallel DBMS VS. MapReduce

With the rapid growth of data, large-scale data analysis and processing is faced a big challenge. MapReduce [32] and parallel database systems [36] are two popular approaches for large-scale data processing.

The debates around MapReduce model and parallel DBMS never stop. At first, David J. DeWitt and Michael Stonebraker published a blog article "MapReduce: A major step backwards" in 2008 [37]. After that, two papers [82, 97] compared the performance of these two approaches. [82] evaluated the performance of the open source Hadoop [116], DBMS-X, and Vertica [111]. It tested these three systems by several benchmark experiments. According to the results, Hadoop costs much less time to load data into the specific file system. However it has poor performance in query execution. Firstly, since data are not required to adhere a kind of schema, it

is impossible/hard to create index for raw data. Secondly, Hadoop bas inefficient processing optimization strategy since the optimizer cannot take advantage of data partitioning information.

Of course, Google people responded the debate and the comparison results strongly in the paper [33] written by Jeffery Dean and Sanjay Ghemawat. In this paper, they argue that there are three flaws in the comparison paper: (1) MapReduce is independent with storage systems and can process data without first loading it into a database. Hadoop can even load and execute queries in the same time that it takes DBMS-X just to load; (2) There are alternatives to reading all of the input data, for example, selecting files based on naming convention or use alternative storage such as BigTable; (3)Many conclusions in the comparison paper were based on implementation and evaluation shortcomings that are not fundamental to the MapReduce model. In addition, they claimed that complicated transformations are often easier to express in MapReduce than in SQL and fine-grained fault tolerance is provided.

In conclusion, both approaches have their own goodness. MapReduce systems are easy to use, fast to load data and support fine-grained fault tolerance. On the other hand, parallel DBMSs support powerful declarative language (SQL) and have better performance due to database technologies such as data partitioning, indexing and query optimization. Then a nature idea is to integrate these two approaches to obtain both advantages. In the following sections, we introduce works on the hybrids of these two approaches.

3.1.2 MapReduce with Hive-level Languages

The MapReduce framework provides simple programming model that users only need to code their own map and reduce functions for their applications. While it is highly flexible in programming applications with this low-level hand coding, it increases the difficulty in the debugging of programs [101] and adds extra burden to the programming beginners [37]. Recently, many works are proposed to integrate high-level languages with MapReduce. Firstly, simpler procedural languages are built on top of MapReduce framework specifically Hadoop, such as Sawzall [83] and Pig [44, 78]. However, they are more suitable for experienced analysts who are familiar with a procedural programming style. Secondly, SQL-like declarative languages are supported by MapReduce frameworks with optimizations to some extent, such as Hive [106], Scope [15], HadoopDB [2], Tenzing [17], Cheetah [21]. As analysis community is very comfortable with SQL, these systems are popular in this field. While these languages significantly improve the productivity of MapReduce programs, MapReduce programs automatically translated from many queries are

often inefficient compared to programs that are manually optimized by experienced programmers. Therefore, more sophisticated translators are proposed to produce more efficient MapReduce programs [69, 49]. Finally, language extensions are built on top of MapReduce frameworks. Such extensions usually have optimizers with special purposes, such as FlumeJava [16] which builds large data-parallel pipelines with relatively simple operations. DryadLINQ [120] provides such extension to Dryad [62] which is another kind of distributed data processing model.

3.1.3 MapReduce with Database Technology

Another criticism for MapReduce frameworks is the slow task execution compared to parallel database systems. [82] shows that parallel DBMSs significantly outperform MapReduce in a variety of tasks. The main reason for the inferior performance of MapReduce is the lack of database technologies, such as data partitioning, data indexing and query optimization. Therefore, many works are focusing on supporting such DBMS features in MapReduce frameworks. The most popular system is HadoopDB [2], recently commercialized by Hadapt. The basic idea behind HadoopDB is that it connects multiple single-node database systems (PostgreSQL [85]) using Hadoop as the task coordinator and network communication layer. It translates HiveQL into MapReduce jobs, some of which are pushed into a single SQL query executed by PostgreSQL while others are executed by Hadoop. With the advantages provided by PostgreSQL, the performance is improved a lot. Rather than using a whole database, some works just integrate a specific database feature with the core MapReduce framework [115, 113]. Hadoop++ [38] provides a non-invasive, DBMS-independent indexing and join techniques to Hadoop to boost its performance without changing its framework at all. Map-Reduce-Merge [22] supports relational algebra primitives and implements several join algorithms efficiently by adding a Merge phase that merges data which has already been partitioned and sorted by map and reduce functions.

3.1.4 Databases with MapReduce feature

Database systems, such as AsterData, GreenPlum, Vertica etc, are extended with some MapReduce capabilities. Greenplum [52] transforms MapReduce code into a query plan which can be executed by the SQL engine on its existing SQL tables. Aster nCluster Database [3, 43] provides users with a procedural API (row and partition methods like map and reduce methods in Hadoop) through which they can implement a UDF in the language of their choice and parallelize the UDF by a MapReduce framework. Teradata's parallel DBMS [117] integrates Hadoop into

its system by offering a fully parallel load utility to load Hadoop data to Teradata. It also allows MapReduce programs to directly read Teradata data through JDBC drivers without any external steps of exporting and loading data to Hadoop. A similar work exists in Vertica [111] which provides VerticaInputFormat [112] implementation that also allows a MapReduce program to directly access data that is stored in Vertica's parallel DBMS, inspired by DBInputFormat [30]. In addition, Osprey [118] implements MapReduce-style fault tolerance in a shared-nothing parallel Database. It divides queries into sub-queries to be executed by PostgreSQL [85], and re-schedules a failed or slow sub-query to a different node which has the replicated data. Similar to MapReduce, Osprey adopts the MapReduce-style load balancing strategy of greedy assignment of work to solve the skew problem.

3.2 Data-intensive Workflows

Workflows are widely used in data-intensive scientific applications since they facilitate the composition of individual executables or scripts, providing an easy-to-use parallelization to domain experts. In this section, we introduce some related works on the general scientific workflows and integrating databases and MapReduce into workflows.

3.2.1 Various Workflow Systems

The basic idea behind a workflow system is that a DAG of coarse-grained jobs with their dependencies are maintained and jobs are executed when its dependencies are met. There are various workflow systems shown in Table. 3.1 [103].

Swift [122] combines a scripting language SwiftScrip for the description of workflows and a runtime system for the efficient parallel execution of jobs in Grid environments. Pegasus [34] and DAGMan [26] can map the workflows to underlying distributed resources efficiently. Both of them use directed acyclic graph (DAG) to describe the workflow. Taverna [77], Triana [104], and Kepler [5] are three popular workflows for scientific problems. They all provide GUI for the composition of workflows with boxes and connectors. The primary components are either web services or Java classes adhering to specific conventions. Another three systems GXP make [103], makeflow [121] and SGE qmake [87] adopt make to describe workflows. There are another two workflow systems *Business Process Execution Language* (BPEL) [67] and *Yet Another Workflow Language (YAWL)* [110] which are widely used by the business community and not described in the Table. 3.1. Similar with Taverna, the primary component of BPEL are Web Services. YAWL is designed with the purpose of being a generic workflow tool to support all kinds of workflow patterns.

	Workflow Description	Primary Component	Target Environment
Swift [122]	SwiftScript	executable	HPC
Dryad [62]	C++	executable	HPC
Xcrypt [57]	Perl dialect	executable	HPC
Hadoop [116]	N/A (fixed)	Java class	HPC
Taverna [77]	GUI	Web Service	WWW
Triana [104]	GUI	Java class	LCS
Kepler [5]	GUI	Java class	LCS
Pwrake [102]	Rake	executable	HPC
GXP make [103]	make	executable	HPC
makeflow [121]	make-like	executable	LCS
SGE qmake [87]	make	executable	HPC
DAGMan [26]	static DAG	executable	LCS
Pegasus [34]	static DAG	executable	LCS

Table 3.1: Various Workflow Systems [103]

As ParaLite aims to express executables in workflows within SQL queries and provide efficient parallel execution for them, we choose GXP make as our workflow engine. Thus, jobs in workflows are presented by SQL queries while the dependencies are described by Make.

3.2.2 Integrating Databases and Workflows

While there is a critical need for workflow systems to manage scientific applications and data and database technology is well-suited to deal with some specific aspects of workflow management, several workflow management systems each of which utilizes database technology to some extent, such as GridDB [70], Zoo [61] and Kepler [5], have been proposed to provide functionality such as simplifying the description of a workflow with SQL queries, improving the performance of the execution and facilitating the management of data.

Some solutions for business workflows focus on the control flow among processes such as active databases [28], relational transducers [1] and enhanced datalog [10]. As scientific jobs are typical long-running and resource-intensive, it is essential to efficiently manage the data-flow. Besides, it is necessary to have sophisticated tools to query and visualize the data. Zoo [61], a desktop experiment management environment built on top of Horse OODBMS, is developed for this purpose. It models the dependencies (relationships) between jobs, input and output data using the

object-oriented language Moose. Job is invoked based on the assigned rules on these relationships. It also provides various workflow auditing capabilities which allow users to use Fox query language to query the state of the database.

GridDB [70] models the inputs and outputs of programs as relational tables. It allows users to define programs and the dependencies between their inputs and outputs with a functional data modeling language (FDM). The execution of programs in the workflow is triggered by the change of the input tables such as insertion of tuples. [75] is another work related to the execution of scientific programs. The work demonstrated the advantages of modern DBMSs such as data indexing, query parallelization and efficient joins by the cluster-finding example from the Sloan Digital Sky Survey. It obtained several times better performance with a database (Microsoft's SQL Server [92]) than previous approach. The work involves invoking program modules from SQL statements which is usually in the form of User-defined Functions (UDF) or stored procedures. As most databases cannot execute executables directly, users who want to invoke such modules as part of SQL statements have to conform to the database specifications while coding and compiling them. They usually have to write them in limited languages (C, C++ and Java). Thus, scientists may face problems in rewriting their applications to run them in a database. Our work falls into this group and concentrates on the expression of jobs in a workflow by SQL queries. Furthermore, it solves the problem of rewriting applications conforming to strict database specifications existing in [75].

While above systems utilize database technology to some extent, [90] provides a tight coupling between data manipulations with databases and workflow managements. It models workflows using a language that is tightly integrated with SQL. Scientific programs in a workflow are associated with active tables or views. The data products are defined in the format of relational tables and the programs are invoked from SQL statements.

3.2.3 Using MapReduce in Workflows

Due to the simple programming model and good scalability across hundreds commodity machines of MapReduce [32] and its open-source implementation Hadoop [116], more and more efforts are made to enable or utilize them in scientific workflows.

Firstly, MapReduce or similar programming models are supported in scientific workflow systems. [42] proposes a MapReduce-enabled scientific workflow composition framework. It designs a set of dataflow constructs, such as Map, Reduce, Loop, and Conditional, and their composition semantics and supports the MapReduce-style parallelization of jobs. Martlet [48] introduces a programming model that

abstracts the parallelization of the computation with *foldr*, *foldl*, and tree functional programming constructs. With these constructs, Martlet is able to extract the complexity of creating parallel processes over underlying distributed data and computing resources, releasing the burden for users from the parallelization implementation.

Then, Hadoop is started to be integrated into workflow systems (e.g. Cascading [14] and Oozie [80]). Oozie [80], an open source workflow system from Yahoo!, is designed specifically for MapReduce (Hadoop) jobs. It organizes MapReduce jobs with action nodes in DAGs and the dependencies between them are represented by the dependencies in DAGs. The functions for flow control in Oozie can be completed by the decision nodes such as fork and join. [123] proposes a strategy to transform XML data processing pipelines to a set of MapReduce jobs which are executed by Hadoop for efficient execution. MRGIS [20] proposes a parallel computing platform based on Hadoop to efficiently execute script-based geoinformatics applications. The Kepler scientific workflow system supports MapReduce workflow composition and management [114]. It provides a general MapReduce actor where Map/Reduce functions are easily expressed by sub-workflows which can be executed in Hadoop. [74] proposes a workflow system to integrate orchestrating MapReduce jobs and structure for data-intensive workflows. It provides C++ API to create DAG, where each job dispatched to Hadoop for parallel execution by a job scheduler. Nova [79] is a continuous workflows on top of Pig [44] for stateful incremental processing. Each module in a workflow is written in Pig Latin [78] and executed on Hadoop.

3.3 Integration of External Executable

To allow integration of data processing methods into query execution plans, relational database systems support user-defined functions (UDF) [94, 13, 95] which are longstanding database features for database extensibility. There are significant research focusing on efficiently using UDFs within database queries in terms of both optimization and execution, such as [19, 55, 56]. However, most of their work in the context of single-node database systems rather than parallel databases for the parallel execution.

There are some works related to the parallelization of user-defined aggregates, table operators [25] and scalar functions [64]. By specifying local and global finalize functions, conventional user-defined aggregates can be executed in parallel [64]. Furthermore, [25] proposes user-defined table operators which use relations as both input and output. The idea of a user-defined table function is supported in most commercial databases including [81, 25, 29]. To enable parallelism and tell the

system the usable of the operator, user-defined table operators require the user to specify a partitioning method , which is inconvenient for the user. Besides, they lack flexibility in input/output formats, development language and reusability of code [64, 65].

On the other hand, MapReduce framework with its most popular implementation Hadoop [116] provides the user procedural API (Map and Reduce functions) to customize their own processing logic. However, it requires programming in relatively low-level languages (most commonly C++ or Java) even for very straightforward tasks that would be trivial in SQL. Integrating third-party binaries and ad-hoc scripts need similar efforts just to wrap them. In addition, Hadoop programs are often slower than equivalent SQL queries because the former lack data indexing and require multiple MapReduce jobs each accessing files for intermediate results.

Therefore, many hybrids of relational databases and MapReduce have been proposed. SQL/MapReduce [43], a part of Aster nCluster Database, proposes an approach to polymorphic user-defined functions providing users with a procedural API (row and partition methods like map and reduce methods in Hadoop) through which they can implement a UDF in the language of their choice. These user-defined methods are parallelized by MapReduce. However, the user still needs to write programs conforming to database APIs and thus cannot directly use existing file-based applications. Hive [106] and Pig [44] add SQL-like functions to MapReduce model, but with a syntax different from SQL. Hive supports queries expressed in a SQL-like declarative language, HiveQL, which are compiled into map-reduce jobs executed on Hadoop. It allows the user to embed custom scripts (mapper and reducer) written in any language. It assumes they read/write data from/to their standard input/outputs and does not accommodate programs that insist on reading data from files. Pig also offers a high-level data manipulation language, which can be assembled in an explicit data flow and interacted with custom MapReduce-style executables. Pig introduces a significant change in its syntax. HadoopDB [2] connects multiple single-node database systems (PostgreSQL [85]) using Hadoop as the task coordinator and network communication layer. The performance is improved a lot but the interface still has limitations since the query planner extended Hive.

3.4 Fault Tolerance

Most of commercial parallel database systems [81, 111, 52, 73] provide fault-tolerance through replication [47, 8, 24, 59]. As they cannot handle the intra-query fault tolerance, if a failure occurs during a long running query, the entire query must be aborted and restarted from the beginning. Such transaction-level fault tolerance

[50] is reasonable for queries with OLTP workload as almost all transactions must be completed within a small amount of time. However, for long-running queries of OLAP workloads, it is costly to restart the query from the beginning as lots of work are lost.

To efficiently recovery a long-running query from the middle of the execution, some researches on intra-query fault tolerance are given. FTOpt [108] provides a intra-query fault tolerance framework which enables the mixing and matching of different fault-tolerance techniques in a single pipelined query plan. However, FTOpt focuses on non-blocking query plans, where data are pipelined from one operator to the next, producing results incrementally. In this case, they assume that aggregation operators, if any, appear only at the top of a plan. Besides, FTOpt uses a brute-force algorithm to enumerate through the search space to get an optimal combinations of fault-tolerance strategies. For a given query plan with n operators, the algorithm runs in at least $O(3^n)$ time. Another work, Osprey [118], provides the ability to detect and recover from failures (or slow nodes) in long-running queries. It divides queries into sub-queries to be executed by PostgreSQL [85], and re-schedules a sub-query which failed or progresses too slowly to a different node based on data replication. Similar to MapReduce, Osprey adopts the MapReduce-style load balancing strategy of greedy assignment of work to solve the skew problem. However, Osprey is designed for data-warehouse applications in which tables are arranged in a star schema and it cannot support many other queries such as non-star joins, nested queries. On the other hand, ParaLite is designed for more general queries.

In contrast to parallel database systems, MapReduce model [32, 116, 106, 2, 44] provides fine-grained fault tolerance by storing all intermediate results to a durable storage before making further progresses. As a result, the Map/Reduce jobs running on a failed worker are rescheduled on another worker, allowing that task to complete without restarting. However, this naive strategy is not always efficient especially for short to middle running jobs. To improve the performance, recent work [23] introduced the ability to partly pipeline data in Hadoop. Another popular lower-level computing platform, Dryad [62], is proposed by Microsoft for data-parallel applications which organized as a data-flow graph with arbitrary computational vertices and communication edges. Hyracks [11] built on top of Dryad allows users to express a computation as a DAG of data operators and connectors. Operators operate on partitions of input data and produce partitions of output data, while connectors repartition the output of operators to make the newly produced partitions available at the consuming operators. Hyracks is trying to explore a more selective fault-tolerance than the naive strategy in Hadoop to achieve the same degree of fault tolerance while doing less work along the way. However, by the time of the publi-

cation of this paper, Hyracks just simply restarts all operators in the same pipeline path with the failed one since data are pipelined between different operators.

3.5 Our Goal

While database systems meet the critical requirement for workflows to efficiently manage scientific data, our goal is to built workflows on top of a parallel database system ParaLite as shown in Table. 3.2. Firstly, ParaLite inherits all advantages of conventional database systems for workflows. All data are stored as relational tables, enabling easily indexing on data and efficiently analysis of data. Jobs are represented by simple but powerful SQL queries, simpling the description of the workflow. Apparently, it should provide high performance for the execution of workflows by taking advantage of database technologies, such as data partitioning, data indexing and query optimizations. Secondly, as each job is a typical individually developed executable, ParaLite is able to provide straightforward expression within a SQL query for such executable. Finally, for long-running jobs, similar with MapReduce, ParaLite can support fine-grained fault tolerance, enabling the recovery from the middle of the execution rather than the beginning to reduce the overall completion time once a failure occurs.

	File-based workflow	MapReduce-based workflow	Database-based workflow	ParaLite-based workflow
Data Storage	File System	HDFS	Database	Database
Description	Low-level	Low-level	High-level (SQL)	High-level (SQL)
Express executables	Easy	Difficult	Difficult	Easy
Performance	Hard to tuning	Worse than database	High	High
Fault Tolerance	Workflow-level	Fine-grained	Transactional-level	Fine-grained

Table 3.2: The Goal of ParaLite

Chapter 4

System Design

Contents

4.1 Architecture

ParaLite is a shared-nothing parallel database system based on a popular single-node database SQLite [93]. The basic idea of ParaLite is to provide a coordination layer to glue many SQLite instances together, and parallelize an SQL query across them. The architecture of our system is shown in Fig 4.1. It uses classic master/worker pattern to organize resources. ParaLite is designed to be a serverless and zero-configuration system, so no process is running before a query is executed. ParaLite has multiple clients which present an SQL interface to users and allows a group of queries to be submitted at the same time.

The master is responsible for transforming received queries into the logical plan (a DAG of operators) which is the key structure to connect each logical component, creating processes for operators on data nodes and scheduling and dispatching jobs to corresponding processes. Data are transferred among data nodes and computing clients, thus the master works only for the controlling decisions and is not a bottle-neck for any data transfer. Each process on a data node receives data from another, handles them using its own processing logic (e.g. join, sort and aggregate) and sends the output data to the next process. The root operator in the logical plan returns results to the clients.

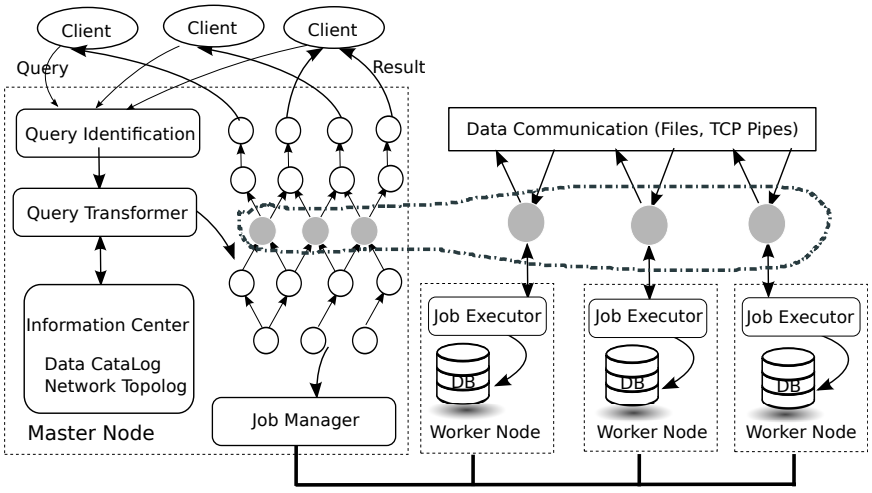

Figure 4.1: Architecture of ParaLite

4.2 Data Model

For a cluster of n nodes, a relation is divided into n partitions either in round-robin fashion or hash fashion. Each partition is then segmented into chunks using the same fashion with first-level partitioning on each node and stored in SQLite database. For instance, relation `A` is partitioned on attribute `A.a` and relation `B` is partitioned on attribute `B.b` using the same hash function with `A`. Then the records from both relations whose values of attributes `A.a` and `B.b` must fall into the same chunk. As a chunk is presented as a single database file of SQLite, a join operation can be simply finished by issuing a query to the database.

Chunks are replicated once using `Interleaved Declustering` which works as follows:

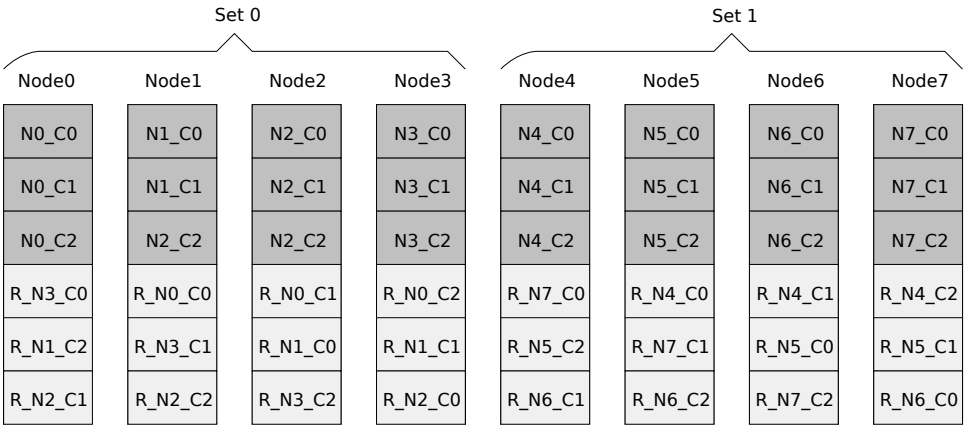

Figure 4.2: Data Model

The whole cluster is divided into several sub-clusters, each of size N , e.g., in Fig 4.2, $N = 4$. Each node has multiple chunks and chunk is the smallest unit for a relation, which means that once a chunk is created, it cannot be divided further. At all times two copies of a chunk exist, called primary copy and backup copy and both copies are located on the same sub-cluster. For each node, all primary chunks reside locally and the backup copies for them are stored on the remaining $N - 1$ nodes respectively. Once a node fails, for instance of node 1 in Fig 4.2, the system reads data from the surviving $N-1$ nodes. Obviously, the load is best balanced when the number of chunks on each node (denoted by M) is times of sub-cluster size N minus one, that is, $M = k \times (N - 1)$. The larger the sub-cluster size N, the smaller is the imbalance in the workload in case of node failure. However, as the sub-cluster size increases, so does the probability of two failures in the same sub-cluster. Failures of two nodes in the same sub-cluster lead to data unavailability. Therefore, we should set the size of sub-cluster according to the structure of the physical resources.

4.3 Query Model

A query is expressed by a DAG (query plan) of relational operators, such as selection, join, aggregation, each of which forwards data tuples to the next operator.

The transformation occurs along with the following chain:
(1) Syntax Parser: It translates a query into a abstract syntax tree consisted of keyword tokens based on SQL grammar. At the same time, it analyzes the query semantically through the interaction with information center which stores information about data partitioning, table attributes, resource usage situation and so on.
(2) Logical Planner: It transforms the syntax tree to logical plan composed of relational operators. Each operator is an executable or a sub-query.
(3) Plan Optimizer: It re-constructs the query plan to be more efficient based on the following rules.

- WHERE-RULE: Filter (where) operator applies some arithmetic calculation to data and gets the satisfied data. In most of the cases, it reduces a large amount of data. Hence, we push these where operators as close as possible to data to reduce the data transfer.

- SPLIT-RULE: We cannot directly send the SQL query to each worker to execute due to the *join* and *group* operators. For example, if the key to be joined is different from the data partitioned key, we surely cannot get correct results. So SPLIT-RULE is used for join and group operators to repartition the data based on appropriate keys. Two special operators, splitter and merger,

are inserted into logical plan before each join and group operator. Splitter is responsible for splitting the output of an operator into several stream based on the join key while merger merges the several input streams before the next operator executes.

- SUB-SQL-RULE: As ParaLite stores data in SQLite databases, it uses SQL query to access data. To take advantage of database technologies of SQLite, e.g. indexing and query optimization, we push as much operations as possible into the query to the underlying SQLite. We retrieve the logical plan from bottom to up and put all operators into a single SQL until a splitter operator encounters.

For example, the query plan of Query 3 of TPC-H benchmark [107] is shown in Fig 4.3. As data are always accessed by SQLite, the leaf nodes of the plan are sub-query operators which read relations using corresponding predicates and produce a row-and-column subset of the relational table. For example, the operator *S7* reads data from relation *Orders* with the query:

```
select orderdate, shippriority, custkey, orderkey
from Orders where orderdate < '1995-10-11'
```

Operators *J3* and *J4* join two relations or counterparts. The output from *J3* is then aggregated and sorted. Specially, some operators can be integrated into a single query if no repartitioning operations are needed. For example, if relations *Customer* and *Orders* are both hash-partitioned across data nodes by the join attribute, *J4*, *S6* and *S7* can be integrated into a single query:

```
select O.orderdate, O.shippriority, O.orderkey
from Orders O, Customer C
where O.orderdate < '1995-10-11' and C.mktsegment = 'AUTOMOBILE'
and C.custkey = O.custkey
```

Each operator is either a *pipeline* operator, which can process each tuple independently without the knowledge of all tuples, or a *blocking* operator which must receive all tuples before emitting the result, e.g, an aggregation operator and a sorting operator. For hash joins, we need to have all tuples from at least one table to build the hash table for it. Then we could in principle emit outputs as tuples

```
select
   L.orderkey,
   sum(L.extendedprice * (1 - L.discount)) as revenue,
   O.orderdate, O.shippriority
from
   Customer C, Orders O, LineItem L
where
   C.mktsegment = 'AUTOMOBILE'
   and C.custkey = O.custkey
   and L.orderkey = O.orderkey
   and O.orderdate < '1995-10-11'
   and L.shipdate > '1995-10-11'
group by
   L.orderkey, O.orderdate, O.shippriority
order by
   revenue DECS, O.orderdate
```

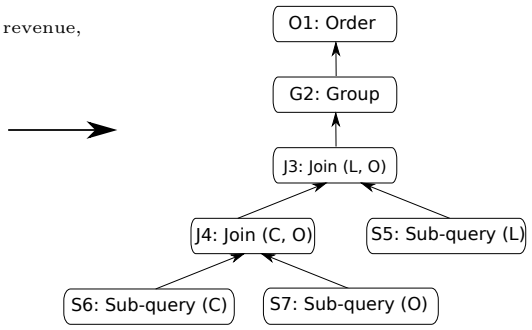

Figure 4.3: A Logical Plan for TPC-H Query 3

from the other table are coming. To simplify the implementation, we consider a join operator as a blocking operator in our work.

Each operator is split into multiple logical tasks and assigned to a set of processors. The number of tasks is determined by the number of partitions for the input tuples of the operator and is usually much larger than the number of assigned processors. If an operator's successor is a pipeline operator, it forwards the output tuples of each task to the its successor as soon as the task is finished. The target processor is chosen based on its processing capacity in terms of estimated runtime. For an operator whose successor is a blocking operator, it holds all output data on memory until it reaches a threshold, at which point it writes them into an intermediate file. Tasks of a blocking operator is scheduled to processors using a greedy algorithm to balance the load across all processors. Once a processor becomes idle, a task is allocated to the processor.

4.4 Execution Flow

The straightforward execution flow of performing a query in ParaLite is shown in Fig. 4.4. Once a query is issued, it is executed through 4 stages as follows:

- Master Creation. As ParaLite aims to be a serverless system, the master is started after a query is issued. ParaLite allows multiple clients to issue queries at the same time. Thus the problem is how to decide a client who is responsible for starting the master and then to let all other clients know the information of the master. ParaLite provides several methods to exchange the information between clients. For users who develop their applications on a shared file system, a database or a file that shared by all clients with exclusion write operation is helpful. If no shared file system is used, users can directly specify the address and port information of the master. The information including

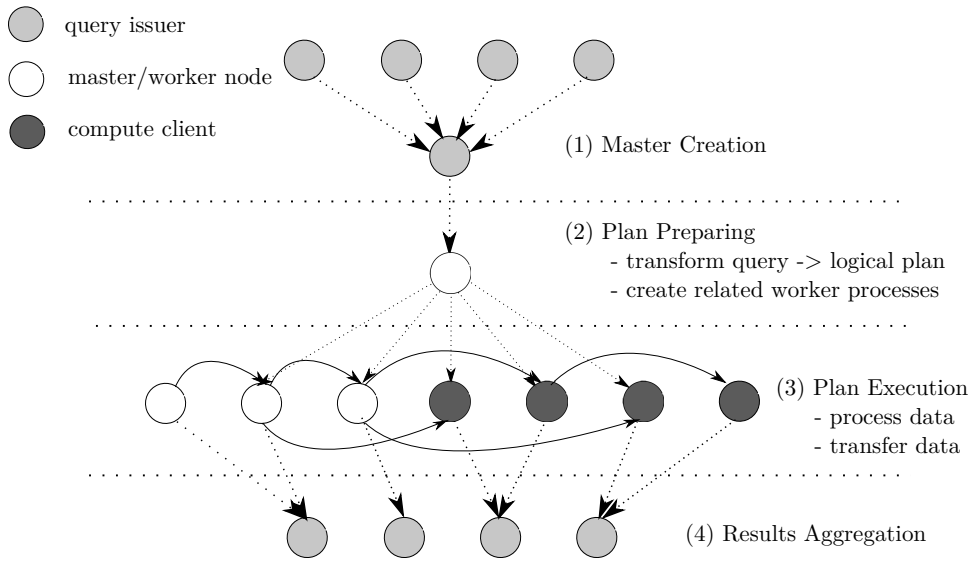

Figure 4.4: The Execution Flow for a Query

the path of the shared database and file and address of the master is specified within a SQL query with a unified interface.

- Plan Preparing. Each client registers the query to the master after it is started. The master collects the clients information and transforms the query into a logical plan of operators each of which is an executable. The master spawns all related executables based on the logical plan. For general SQL operators such as join, aggregation and selection, their processes run on the data nodes while the processes for the user-defined executables introduced in Section 5.1 are started on computing clients. Then the master splits each operator into a set of logical tasks (the number of tasks can be specified by the user) which are scheduled to corresponding processes according to the dependencies of operators in the logical plan, using a greedy manner.

- Plan Execution. Each operator process presents a channel through a TCP pipe for receiving the control message from the master and tuples from other processes. After processing received data, the process forwards the result tuples to another process. The result tuples are split with respect to the number of tasks assigned to operators. For example, if a join operator $J1$ has N tasks, the output tuples of its children operators are partitioned by N chunks. Specially, for the sub-query operator, it has the tasks as many as all chunks that the original data for the related table is partitioned to. Each process handles a task at a time. If the operator is decided to be checkpointed, the result tuples are written to the local files. After all tasks are finished, all

processes are destroyed.

- Results Aggregation. The output tuples from the top operator of the logical plan are sent to the clients who issue the queries. If only one client exists, the results are aggregated and returned to it. If there are multiple clients, the results are distributed to them randomly.

4.5 Easy-to-use Features

ParaLite is designed to be a lightweight system which provides easy-to-use features to users:

- Server-less: There is no daemon started before queries' execution. ParaLite inherits this feature of the underlying SQLite database system. All processes are spawned according to the logical plan of the query and destructed after the query is finished.
- Zero-configuration: Necessary configurations, such as the addresses of the master nodes and data nodes, are specified within a query. ParaLite also provides some performance tuning configurations, such as the PRAGMA statement which is used to modify the operation of the SQLite library or to query the SQLite library for internal intermediate data, the number of processes for each operator and the size of a block that is executed by an executable at a time (described in Section 5). The user can specify all these options in a configuration file or leave them as default. ParaLite assigns these parameters with the best values by default.
- Fast job start-up: Although spawning all related processes after receiving the query takes more time than keeping them running for the time, this overhead is much smaller than that in Hadoop.
- Independent database files: The data files behind ParaLite are a set of SQLite database files. As SQLite stores a database into a single file, it is easy to manage database files, such as replicating and migrating. It is also convenient to port SQLite databases to ParaLite databases simply by updating the metadata information of ParaLite.

4.6 Workflows with ParaLite

ParaLite aims to facilitate the description of workflows with SQL queries and the management of data. As Fig. 4.5 shows, to develop a workflow on top of ParaLite,

the input data are firstly loaded into ParaLite as relational tables, and then each job expressed with SQL query performs corresponding computations on related tables and creates another table to store its output data.

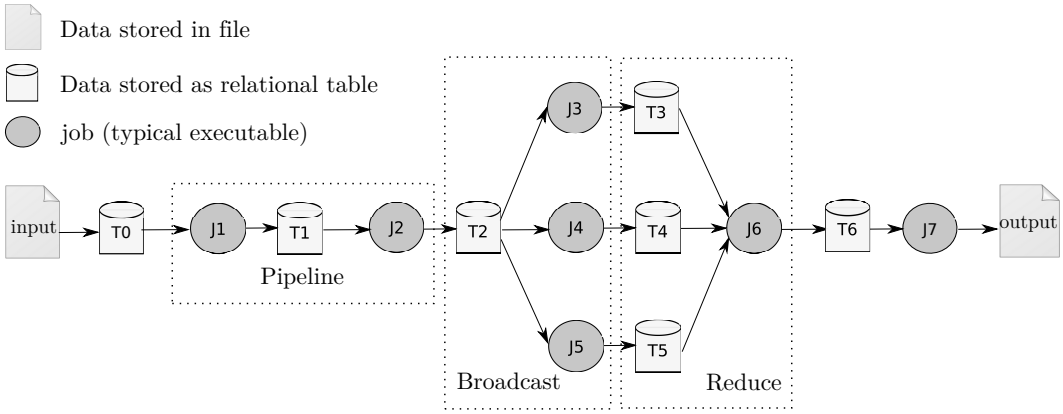

Figure 4.5: An Example of Workflow on top of ParaLite

Firstly, SQL queries are well-suited to describe workflows for the common data access patterns:

- Pipeline: It is a chain of a set of jobs where the output of one job is the input of the next job. The pipeline pattern could be expressed with either multiple queries (each job is expressed with a single query) or a single query which contains a chained UDXes (introduced in the next chapter) if each job is an executable. While the former method provides independent view of the execution of each job, it has to store the output of each job into the database, wasting much time on the data loading process. The latter executes jobs and data are pipelined from one to another.

- Broadcast: A single input is processed by a number of jobs at the same time. The broadcast pattern is easily expressed with several queries that retrieve the same relational table.

- Reduce: The input for a single job is produced by multiple different jobs. For example, a job checks the results of previous jobs for a convergence criterion, and a job that calculates the summary statistics from the output of multiple jobs. An executable working on joined data from several input tables is enough for the Reduce pattern.

Secondly, it is natural to simplify the management of data including all intermediate data with ParaLite. It supports indexing on data, making selection of a

subset of data efficient. This feature is especially useful for many natural language processing applications which end with a search engine on indexed data, e.g. the MEDLINE to MEDIE Indexing workflow [71] which creates indexing database for a running search engine called MEDIE.

Finally, ParaLite provides transparent parallelization of jobs and improves the performance of the workflow: 1), data are naturally partitioned on many data nodes eliminating explicit big file split; 2), typical SQL tasks such as selection, join and aggregation are processed by a set of data nodes in parallel; 3) with support of UDX and collective query, ParaLite provides transparent parallelization of UDX across multiple computing clients with optimized data transfer from data sources to clients. The performance is easily tuned by specifying different block size based on the characteristic of the executable.

Chapter 5

Integration of Executable

Contents

5.1 User-defined Executable(UDX)

As the intended applications for ParaLite are workflows typically built out of various independently developed executables and scripts, ParaLite extends SQL to support arbitrary executable called User-Defined Executable (UDX).

5.1.1 Syntax of UDX

A ParaLite UDX is an executable file which can be written in any language. This is very flexible because a user does not need to develop a program respecting to rigid formatting rules such as <key, value> input/output format or write code according to pre-defined procedural methods. User can use arbitrary format of input and output and any programming language to implement their functions. The only condition is just to make sure the program is executable. This has been very useful for data-intensive science computations such as a linear algebra package for solving linear equations and a natural language processing library. In these applications, most functions are developed by domain experts and then reused by others on diverse workflows. ParaLite programs allow such functions to be reused without changing any code.

The syntax of UDX shown below is similar to that of traditional User-Defined Function. Firstly, the name of an UDX could be a random string e.g *X* in the example. Secondly, an UDX can work on and produce arbitrary columns. It extends *AS* syntax a little bit so that multiple columns are allowed to be the output of a UDX by using "*AS (col1, col2,...)*". Finally, the definition of an UDX provides flexible input/output format with a set of options such as, *input*, *input_ row_ delimiter*, *output* and *out_ row_ delimiter*. The options related to input allow the system to correctly extract and organize data for the executable while output-related options tell the system how to parse the output of the executable and store them in a relational table. Specially, *input* and *output* options can specify that the input/output for the executable comes/goes from/to the standard input/output or files. This ability is especially useful for file-based programs commonly existed in NLP applications. In addition, to avoid create and compile an UDX before the query is executed, ParaLite allows users to define it within the query using *WITH* clause. It starts from a command line followed by data format options mentioned above.

```
select col1, X(col2) as new_col2
from T
where <predicate1, predicate2, ...>
with X = "command_line"
```

5.1.2 Examples

In this section, we take some examples to elaborate the usage of UDX. First of all, let's define the schema for a table *DOCUMENT*:

```
paper_id | title | author | year | text
```

- Grep Task

  Grep task is considered as a typical MapReduce task which scans through a large set of records looking for a three-character pattern. This task can be expressed by a simple SQL query:

  ```
  select * from DOCUMENT where text like '%XYZ%'
  ```

  It is also easily performed by a query with shell scrip command "grep" as a UDX:

  ```
  select F(text) from DOCUMENT with F = "grep XYZ"
  ```

All data of column `text` retrieved from table `DOCUMENT` is processed by the UDX `grep XYZ` and the filtered data are returned.

- Word Count Task

  Word count task is also a typical MapReduce task[32] to count the number of occurrences of words in a large collection of documents. This task processes a document table in which a single row is a single document with its descriptions and generates a word table in which a single row is a word with its occurrences. While this task could be easily expressed by MapReduce researcher with a Map and a Reduce job, there is no easy way to perform it in database community unless the big text could be split into words. With UDX, it is straightforward to integrate text splitter into general group by SQL task to calculate the count for each word.

```
select word, count(*) from (
   select F(text) as word from DOCUMENT
   with F = "awk '{for(i=1;i<=NF;i++) print $i}'"
)
group by word
```

  Column *text* is a long article split into independent words by an *awk* command in the nested SQL. The occurrences for each word is simply counted by grouping words from the output of the nested query. The command *awk* reads data from standard input and writes results to standard output.

- Sentence Split Task

  The sentence split task is to split a big text into sentences and it is normally the first step of almost all text-processing applications. It involves a third-party binary *geniass*[45] developed by domain researchers which reads a text from file, splits it into sentences by inserting line breaks between sentences within a paragraph and empty lines before the sentence from another paragraph, and finally outputs the sentences to a file.

```
select paper_id, F(text) as sentence
from DOCUMENT
with F = "geniass" input 'src_file' output 'dest_file'
                   output_row_delimiter EMPTY_LINE
```

In the query above, the user needs to specify the input and output for the executable to be a file. This is different from the word count task in which the executable reads/writes data from/to the standard input/output. Besides, if the option *output_record_delimiter* is empty line, it means that all results between two empty lines belong to a single record. So the results for the query is as follow:

```
   paper1 | sentence1
            sentence2
-------------------------
   paper2 | sentence1
            sentence2
-------------------------
          ...
```

Each result record has two columns of *paper_id* and *sentence*. The second column contains multiple sentences separated by line breaks. By setting the *output_record_delimiter* option to be *NEW_LINE* as described below, each line of the result from the executable becomes a single record.

```
select paper_id, F(text) as sentence
from DOCUMENT
with F = "geniass" input 'src_file' output 'dest_file'
                   output_row_delimiter NEW_LINE
                   output_record_delimiter EMPTY_LINE
```

The query above produces a result table still with the two columns. However, the second column of each record only contains one sentence.

```
    paper1 | sentence1
-------------------------
    paper1 | sentence2
-------------------------
    paper2 | sentence1
-------------------------
    paper2 | sentence2
-------------------------
           ...
```

Now, we write another program *geniass_ wrap* which reads text from standard input and writes sentences with its ID (separated by '\t') to the standard output. So a user should specify the *output_ col_ delimiter* to let database system know how to convert data into the right schema as described in the following query.

```
select paper_id, F(text) as (SID, sentence)
from DOCUMENT
with F = "geniass_wrap" output_col_delimiter '\t'
                        output_record_delimiter EMPTY_LINE
```

As a result, the query returns tuples with three attrbutes: *paper_ id*, *sentence_ id* and *sentence*.

```
    paper1 | 1 | sentence1
-------------------------------
    paper1 | 2 | sentence2
-------------------------------
    paper2 | 1 | sentence1
-------------------------------
    paper2 | 2 | sentence2
-------------------------------
```

- Sentence Parser Task

  Sentence Parser Task is to get the relationship between each word using a NLP tool *enju*[39] which is a fast, accurate, and deep parser for English texts. The task firstly splits the text into sentences using *geniass* and then applies *enju* program to each sentence. This simple workflow can be expressed with two nested UDXes as follow.

```
select E(F(text)) as enju_result
from DOCUMENT
with F = "geniass" input 'src_file' output 'dest_file'
     E = "enju" output_row_delimiter EMPTY_LINE
```

As the output format of *geniass* is the same with that of the input of *enju*, the specifications of them can be ignored in their definitions. The outer UDX *enju* tells the system that the result records are separated by empty line with the corresponding option. Each of the result records only has one column *enju_result* with multiple lines.

5.2 Execution Model of UDX

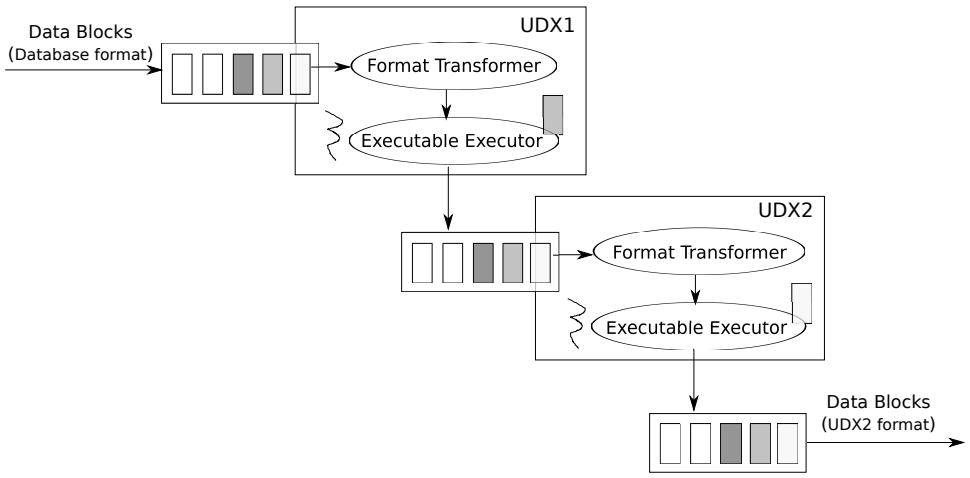

Figure 5.1: The Execution Model of UDX

An instance is initiated for a program when a client issues a collective query. As Fig. 5.1 shows, inside the instance, the execution of the executable uses an event-based programming style. Each instance has an incoming queue for storing the received blocks of data (splitting data into blocks is introduced in Section 5.3.2). In our execution model, only one thread is assigned to each incoming queue because the main work of the thread is to spawn a process for the execution of the executable. We assume that the user controls the parallelism degree in terms of the number of computing clients based on their knowledge of the characteristic of the executable. The instance has to publish a method to receive data using a pipe, sockets or files. Once a block of data is pushed into incoming queue and no thread is running for this instance, the instance starts to process the block using the executable in a new thread. It firstly transforms the data into the correct input format of the executable based on the output-related options of its previous executable (or the data format in the database) and the input-related options in its definition. The converted input is then fed to the executable either from standard input or a target file. After the execution, the results are pushed to the outgoing queue of this instance which is

also an incoming queue of its successor instance if it exists. Besides, since data are sent to the instance based on the processing state dynamically, the instance is able to obtain the real-time execution status of the program.

If a query has nested UDXes, ParaLite supports the pipelined executions of multiple executables also as Fig. 5.1 shows. After a data block is processed by the first instance, the result data are pushed to its outgoing queue or the incoming queue of the next instance, triggering the execution of the next executable. Meanwhile, the first instance continues to obtain and then process the next block without waiting for the all other executables to complete the previous block.

5.3 Parallel Execution of UDX

5.3.1 Concept of Collective query

To understand collective query better, we first give a brief review of parallel database systems.

A parallel database system provides the same functionality as a centralized DBMS with the ability of transparently distributing data across nodes and parallelizing queries. It typically consists of a single master node and multiple data nodes. A master is responsible for receiving a query from a client, converting the query to a parallel execution plan, scheduling the plan to worker nodes, and assembling the individual "pieces" of the final results up into a single result set to the client. As such, the master node hides the distributed nature of the system and presents users a single system image. Data nodes provide the data storage and the query processing backbone of the appliance. All permanent data are partitioned and stored on each data node. All queries, therefore, must access data stored in DBMS on some (or all) of the data nodes. A data node is responsible for processing one or more steps of the execution plan. The intermediate result data are transferred among them directly without passing them to the master node.

Let us assume a user wants to apply a parallel processing on the data accessed from a parallel database such as loading the data into another system (e.g. MapReduce system) or performing a further analysis which cannot be expressed within a SQL query. In this case, a traditional approach is that a user issues a query, gets all result data from the database system and then distributes them to multiple clients who perform the further processing on the result data in parallel. As Fig. 5.2 shows, in a parallel database system, data are firstly partitioned across multiple data nodes. After receiving a query from a client, all result data are integrated and returned to the single client. Then to enable the parallel processing on the result data, the client distributes them across multiple computing clients.

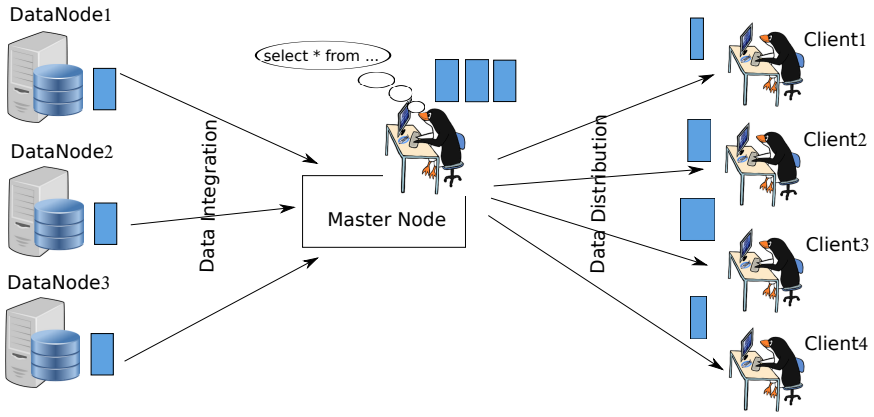

Figure 5.2: Data Transfer in Conventional Parallel DBMS

Obviously, the single query issuer can easily become a bottleneck. Moreover, as the approach prohibits us to take the advantage of co-allocating computing clients with data in the database, data transfer probably brings much unnecessary overhead to the overall execution. For example, if the computing clients are located on the same physical data nodes, it is straightforward and efficient that each client gets data locally from the corresponding data node, instead of integrating data from data nodes first and then distributing them to the same nodes again.

Hence, to solve this problem, we propose the concept of *collective query* to take advantage of co-allocation of parallel compute clients and data sources. A collective query is a single query issued by many clients who collectively receive the results of the query. As Fig. 5.3 shows, a set of clients issue the same query to the database system. They just notify the system to get a part of data and don't care about what exact data they will obtain. The system performs the query received from the clients and distributes the result to them. As the system knows the destinations of the result in advance, it allows the data transfer directly from the data nodes to the clients without passing through the master. In this case, data are not required to be integrated first and distributed later. The best situation is that when a computing client is running on each data node and data are already balanced among them, no data are transferred between nodes at all and each client gets data locally from the data node the client is running on.

ParaLite uses the concept of collective query for the parallelization of executables. The query issued by multiple clients contains one or more UDX(es). The results of the query are distributed to the clients and then processed by them in parallel, using the executables defined in the query.

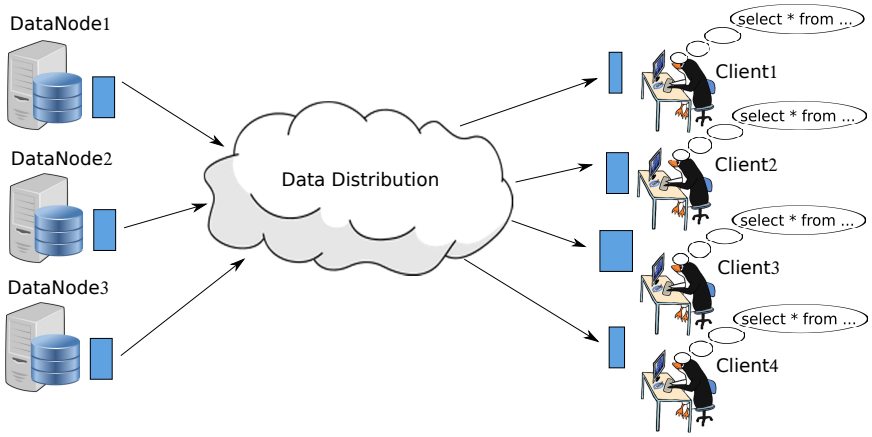

Figure 5.3: Data Transfer in ParaLite

5.3.2 Data Distribution

The key issue for collective query is to efficiently distribute data to clients from data nodes. The result data are always stored in multiple nodes, thus, a more general problem setting should be that given data on N data nodes and m clients, to determine which data should go to which client. In data intensive applications, a client may perform a significant amount of computation, therefore, ParaLite should consider not only communication cost but also computation cost.

ParaLite implements two-phased DLLB (Data Locality with taking Load Balance into consideration) algorithm to solve the problem. The first phase in the algorithm is to generate tasks on each data node by splitting data into small blocks. Each block size is denoted by *bsize*. A block is the smallest unit that is transferred to a computing client at a time and a process of the executable is spawned for a block of data. Invoking a process on a block of data rather than a single tuple reduces the start-up overhead a lot for most NLP programs.

Once splitting is complete and execution has begun, we enter the load-balancing phase. When a client i on the node A becomes idle, a task (block) must be transferred to it from some data node. Our goal is to balance load across all clients but with the smallest data transfer cost. Thus, the target data node T should be:

(1) node A, if A is a data node, or,

(2) node B, who has the maximum expected completion time:

$$B = \underset{J}{\text{argmax}}\ ECT_J$$

ECT is an expected completion time as measure of the load of a node. ECT of a data node J relies on the total data and clients it holds and can be calculated as

(assuming c clients are running on it):

$$ECT_J = lsize_J / \sum_{i=1}^{c} s_i + max_{i=1}^{c}(bsize/s_i - stime_i)$$

$lsize_J$: the size of left (unassigned) data on data node J

s_i: the speed of client i

$stime_i$: the time for client i starting to process a task

The speed of each client is initiated as a random number. The ECT should be infinite if a data node has no client on it. Once a client completes a task, it sends an ACK message to the master to notify about its IDLE status and report its processing speed which is used for the next scheduling. Since it is difficult to provide an exact measure of real-time speed for a client, ParaLite always estimates it by the last speed of a client. On the other hand, after the master receives a ACK message from a client, it updates its speed and data information of related data node whose data are processed by this client. Once a client becomes idle, the master firstly calculates the ECT of all data nodes and decides a target data node based on the formula above. The computational complexity for deciding a target data node is $O(n)$ where n is the number of client. It costs 10^{-6} seconds if there is only one client based on some experiments we conducted.

Once the target T is chosen for an idle client i, it must decide how much data (how many blocks) are needed to be transferred to the idle client. DLLB employs the following equation to calculate the number of blocks:

$$n = \begin{cases} 0 & \text{if } T = B \text{ and } ECT_T(cur) < Com(1) + ECT_i(1) \\ 1 & \text{otherwise} \end{cases}$$

Data are transferred to the idle client only when the transfer provides a gain in the completion time, that is, the sum of the task transfer time ($Com(1)$) and the estimated completion time of the task on the idle client ($ECT_i(1)$) should be smaller than the estimated completion time of all tasks on the target node ($ECT_T(cur)$).

The DLLB algorithm is flexible for data scheduling. It first takes data locality as a main consideration and always tries to perform calculation on local data. But if the calculation is CPU-intensive and data transfer can get a gain to the calculation, data are transferred to a remote client to be calculated. At most one block is transferred at a time, which lets the master node have better control on data. Fine-grained data scheduling adds extra overhead of making decision to the master node, but the cost of making decisions is negligible even the data unit to be transferred is 1 compared with the cost of data transferring and calculation.

In addition, DLLB algorithm allows new clients to join at any time before all data are scheduled. When ultra long time is taken and the task is still unfinished, user can

start more clients on more resources by simply issuing the same collective queries. Although it provides big controllability to users, extra burden is also brought to them. So in the future, ParaLite should add more clients automatically by itself in order to finish a task faster without the interference of users.

5.4 UDX Compared to UDF

Compared with ordinary User-Defined Function (UDF) in conventional parallel database systems, our implementation of User-Defined Executable (UDX) has the following advantages:

- UDX doesn't require the user to write any program and register to the database before it is executed. To define UDF in conventional databases, the user has to write a specific program conforming to strict database specifications. This is very troublesome in workflows as typically only executables are provided. In this case, for each executable, a program that encapsulates the invoking of the executable is required. Besides, programs are usually constrained with the language they can use as most database systems only support UDF written in C/C++ or Java.

- The UDX implementation does not invoke the executable on every single tuple while the implementation of UDF does. As typical NLP programs have high start-up overhead, invoking such programs on every single tuple would heavily degrade the overall performance.

- The execution of an UDX is not bound to database nodes and it can be distributed to arbitrary clients for larger scale execution and computational load balancing. This loose coupling with database nodes is also very useful in the case of that data are stored in a set of nodes while the related executable is installed in other sets of nodes for some reasons, e.g. the licenses. On the other hand, an UDF can only be parallelized across data nodes pre-configured before the database server starts.

- UDX parallelization is efficient as it optimizes data transfer between data nodes and computing clients. Most commercial database systems take a naive strategy to parallelize UDF which assigns a whole partition of data to a local processor without consideration of its load. Moreover, the implementation of the UDX allows flexible control on the parallelism degree by increasing or reducing the number of computing clients.

Chapter 6

Intra-query Fault Tolerance

6.1 Problem Setting

Assume that the logical plan of a query is $G =< V, E >$, where V is a set of operators $V = v_1, v_2, ..., v_n$ and E is the set of edges (or dependencies) between operators $E = (v_i, v_j)|v_i, v_j \in V$. Each operator is started only after all of its children operators are finished and multiple operators that don ' t have dependencies with each other can be executed in parallel. When the query is executed without any failure, the runtime time T_{total} of the query is calculated as a function of the operator and edge sets:

$$T_{total} = F(V, E) \tag{6.1}$$

In a failure-prone environment, we assume that each operator is executed with a probability of failure $P = p_1, p_2, ..., p_n$. If an operator v_i is failed, all of its children are required to be re-executed to replay the input for v_i if they are not checkpointed. Otherwise, the checkpointed operators can simply replay its output by reading them from a durable storage. The problem is that what operators should be checkpointed. Not making a necessary checkpoint may lead to a loss of important computation, affecting the overall execution performance while making an unnecessary checkpoint leads to an increase in the checkpointing overhead as well. Let's denote another variable $CK = ck_1, ck_2, ..., ck_n$ to mark if a checkpoint is necessary or not for each operator. In the case of failure, the expected runtime of the query is calculated by the following equation:

$$T_{total} = F(V, E, CK, P) \tag{6.2}$$

Therefore, the problem is equivalent to get the a set of checkpoints to minimize the expected runtime:

$$CK_{opt} = \underset{CK}{\operatorname{argmin}}\ \mathrm{F(V,\ E,\ CK,\ P)} \tag{6.3}$$

6.2 Fault Tolerance Strategy

Generally, data are pipelined from one operator to the next. Once a job fails, all operators are required to be re-executed as all intermediate data are lost. To avoid

operators' re-execution, a common strategy is *checkpointing and rollback recovery approach* which periodically records the state of operators to a durable storage. To ensure the right recovery from a failure, an operator is required to save sufficient information to replay its state such as join hash tables and partial aggregation results. Once failures occur, the operator restarts from the last state (checkpoint). In our work, we take a simpler checkpointing method which does not save internal states of an operator but only saves the output of an operator in a durable storage. This checkpointing method works because the execution of each operator can be divided into many independent parts and only failed parts need to be re-executed upon a failure. For example, a join operator J has two predecessors $S1$ and $S2$. The outputs of $S1$ and $S2$ is partitioned into hundreds of parts (each part is a logical task) by the join attribute and the tasks are then scheduled to a specific number of processors (which is usually much smaller than the number of tasks) performing data joining. Once a failure occurs, saying a processor for J fails, only the failed task (which is running exactly when the failure occurs) is re-executed if the output of J is checkpointed. As a result, the recovery overhead of the failed operator is very limited as each task usually runs in a small time.

With checkpointing strategy, the places to insert checkpointing heavily affect the overall execution performance. On one hand, not making a necessary checkpoint may lead to a loss of important computation, degrading the overall execution performance. A checkpointed operator speeds up the recovery for its successor operator as reproducing the output tuples is simply re-reading materialized data. For example, for a query plan $A \rightarrow B \rightarrow C \rightarrow D$ with only A checkpointed, if D fails, operator B, C and D require to be re-executed while A does not. So only checkpointing A leads to too much work lost when a failure occurs. On the other hand, making unnecessary checkpoints leads to an increase in the checkpointing overhead as well. We also take the above query plan as an example. If the operator C is light on CPU but emits lots of tuples, it is unlikely worth being checkpointed. An extreme case is that the checkpointing time is wasted if no failure occurs.

6.3 Cost Model

In this section, we explain how we model the cost of an execution plan of a query and propose the problem for the selective checkpointing.

Processing Cost: For operator i, the processing cost T_{P_i} is the delay introduced by the operator. T_{P_i} is the sum of execution time for its input tuples and the communication time for its output tuples as shown in Equation 6.4.

$$T_{P_i} = T_{EXE_i} + T_{SEND_i} \tag{6.4}$$

T_{EXE_i}: the cost for executing input tuples of operator i;
T_{SEND_i}: the cost of sending output data to the successor for operator i.

To estimate T_{P_i}, we assume the following two functions are known for each operator i:
$f_i(N_{in\_tuple_i})$: this function provides the number of output tuples produced for a given number of input tuples $N_{in\_tuple_i}$ of operator i:

$$N_{out\_tuple_i} = f_i(N_{in\_tuple_i}) \tag{6.5}$$

$g_i(N_{in\_tuple_i})$: this function provides the time to produce all output tuples for a given number of input tuples $N_{in\_tuple_i}$ of operator i:

$$T_{EXE_i} = g_i(N_{in\_tuple_i}) \tag{6.6}$$

In practice, such infomation could come from profiling and statistics, or could be supplied by user. Specifically, the function f_i is simply given with respect to the selectivity for each operator while it is more complicated to estimate g_i. The function g_i models the processing for each tuple, e.g. for an operator only with a sorting algorithm, the $g_i(n)$ is of the form $n \times log(n)$. We ignore other fixed, auxiliary cost for operators such as initiating and terminating the operation. If the output tuples of the operator i are divided into N_{task} tasks with hash function $h(key) = i$ $1 \leq i \leq N_{task}$, the partitioning cost is added to T_{EXE_i}.

The cost for sending $N_{out\_tuple_i}$ tuples depends on the number of the successor operator. For the blocking successor operator, such as group operator, the output tuples are required to be re-partitioned on the group key and transferred to all successor processors. For other operators, the output tuples are transferred to only one or a few processors.

Checkpointing Cost: We set a checkpoint value CK_i for operator i:

$$CK_i = \begin{cases} 0 & \text{if a checkpoint for } i \text{ is not necessary} \\ 1 & \text{otherwise} \end{cases}$$

For operator i, the checkpointing cost T_{C_i} represents the cost to write the output tuples of i to disk:

$$T_{C_i} = CK_i \times T_{I/O} \times N_{out\_tuple_i} \tag{6.7}$$

$T_{I/O}$: the time taken to write/read a tuple from/to disk.

Recovery Cost: For operator i, the recovery cost T_{R_i} depends on the checkpointing values of both i and its predecessors. The recovery time of an operator i contains two parts as equation 6.8 shows: T'_{R_i} the cost for getting the input tuples again and T''_{R_i} the cost for the re-execution of the operator i itself.

For the former, assuming that the operator i fails, its input tuples are required to be reproduced, that is, all of its predecessor operators need to reproduce their output tuples. Thus, the recovery time for the operator i is determined by the predecessor who takes longer time to reproduce its output tuples as equation 6.9 shows. For each of its predecessors j, the cost for reproducing the output tuples of j is reading data from disk if j is checkpointed ($CK_j = 1$). Otherwise, if j is not checkpointed ($CK_j = 0$), j must be re-executed.

$$T_{R_i} = T'_{R_i} + T''_{R_i} \tag{6.8}$$

$$T'_{R_i} = \max_{j \in PRED_i} ((T_{P_j} + T'_{R_j}) \times (1 - CK_j) + CK_j \times T_{C_j}) \tag{6.9}$$

For the second part of the recovery cost, T''_{R_i}, if the successor of operator i is a blocking operator, i keeps the output tuples of received tasks until all tasks are finished. As a result, all tasks are re-executed if this operator is not checkpointed. If the operator is checkpointed, only the failed task (which is running at the time the operator fails) needs to be re-executed. If the successor of operator i is a pipeline operator, the output of a task is sent to its successor quickly. Thus, still only the failed task needs to be re-executed.

Objective Function: Assuming that infinite resources are used, the expected runtime of an operator i is the sum of the processing cost of the operator T_{P_i} (Eq 6.4), the expected runtime of its children, the recovery cost T_{R_i} (Eq 6.8) and the checkpoint cost T_{C_i} (Eq 6.7).

$$T_i = T_{P_i} + \max_{j \in PRED_i} (T_j) + p_i \times T_{R_i} + T_{C_i} \tag{6.10}$$

$PRED_i$: the set of predecessor operators of i

p_i: the probability of a failure for operator i.

Obviously, the expected runtime of a query plan T_{total} is estimated by the following equation according to equation 6.2 (where operator 0 represents the root of the query plan):

$$T_{total} = F(V, E, CK, P) = T_0 = T_{P_0} + \max_{j \in PRED_0} (T_j) + p_0 \times T_{R_0} + T_{C_0} \tag{6.11}$$

The objective is to find the optimal checkpointing value for each operator to minimize the expected runtime of a given plan.

$$CK_{opt} = \underset{CK}{\operatorname{argmin}}\ T_{total} \tag{6.12}$$

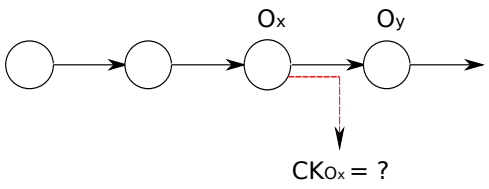

Figure 6.1: Simple plan

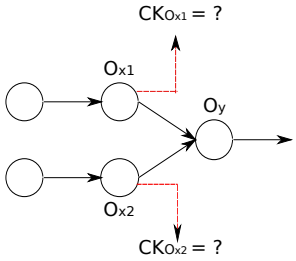

Figure 6.2: Complex Plan

6.4 Heuristic Algorithm

A straightforward approach to the problem is to retrieve all solutions in the full search space and get an optimal one. However, if the size of the query plan (saying that the number of operator is n) is large, this approach is not practical as the search space S_G is 2^n. So we propose a heuristic algorithm (using divide-and-conquer approach) to reduce the search space. Although the algorithm might not produce the global optimal solution, its efficiency is verified by the experiments.

6.4.1 Reduction of Checkpointing Candidates

First of all, we use a simple inequality to filter the operators which should not be checkpointed. Let's firstly consider a simple sub-plan in which each operator has at most one upstream operator as Fig 6.1 shows.

Assume that a failure occurs at the operator y, the completion time of y depends on whether operator x is checkpointed ($CK_x = 1$) or not ($CK_x = 0$). When x is not checkpointed, the recovery should start from the beginning and the completion time of y should be:

$$T_{total_y\{CK_x=0\}} = 2 \times T_{total_x} + T'_{R_y} \tag{6.13}$$

With the checkpoint, the recovery of y starts from reading tuples from disk and the completion time of y with consideration of a checkpoint is:

$$T_{total_y\{CK_x=1\}} = T_{total_x} + 2 \times t_x + T'_{R_y} \tag{6.14}$$

t_x is the I/O cost for reading/writing the output of x from/to the disk. For n tuples, the I/O cost is estimated as $t_x = T_{I/O} \times n$.

To get benefit from making a checkpoint with a failure, the completion time of y with a checkpoint should be smaller than that without a checkpoint:

$$T_{total_y\{CK_x=1\}} < T_{total_y\{CK_x=0\}} \quad \Rightarrow 2 \times t_x < T_{total_x} \tag{6.15}$$

Algorithm 1: singleCheckpointing: algorithm for setting the checkpoint value for an operator without sibling

Input: i: an operator

Output: the checkpointing values of i

if $inequality6.15(i) = True$ **then**
  return 1
else
  return 0

Inequality 6.15 intuitively shows that the process of a query plan can gain from a checkpoint of operator i only if the cost for writing and reading the output of i is smaller than its completion time. This observation generates the opportunity to reduce whole search space by setting $CK_i = 0$ if i does not satisfy the inequality.

Next, let's consider a more complex sub-plan as Fig 6.2 shows. Operator y has two upstream operators x_1 and x_2 which are executed in parallel. As the completion of y is decided by the slower operator, the checkpointing decision of the two operators depends on both execution times. We take the following algorithm to decide the checkpoint values for x_1 and x_2.

Algorithm 2: siblingCheckpointing: Algorithm for setting the checkpoint values for two sibling operators

Input: i and j: two sibling operators

Output: i and j with the checkpointing values

$small \leftarrow min\_execution\_time(i, j)$;
$big \leftarrow max\_execution\_time(i, j)$;
$CK_{small} \leftarrow 0$;
$CK_{big} \leftarrow singleCheckpointing(big)$;
if $CK_{big} = 1$ **then**
  if $T_{P_{big}} + T_{C_{big}} > T_{P_{small}} + T_{C_{small}}$ **then**
    $CK_{small} \leftarrow 1$
  else
    $CK_{small} \leftarrow singleCheckpointing(small)$

Algorithm 2 firstly decides the checkpointing value for the operator with larger execution time (denoted by i) according to the Algorithm 1. The checkpointing value of the operator with smaller execution time (denoted by j) is decided by i's checkpointing value. If i is not worth being checkpointed, the algorithm thinks it is better for not checkpointing j because the recovery time for their parent is decided by the execution time of i. When i is checkpointed, j is also checkpointed if the

overall cost of the processing and checkpointing for j is smaller than that of i and otherwise the checkpointing value of j is set through the Algorithm 1.

6.4.2 Divide-and-Conquer Approach

Divide-and-conquer approach is a common strategy that comes next to the brute-force but reduces the search spaces. It works as follows: query plan G is divided into sub-plans, denoted by $G^{(i)}$, with smaller plan search spaces S_G^i such that the globally-optimal choice in S_G can be found by composing the optimal choices found for each S_G^i. Each sub-plan is the smallest unit within which an optimal checkpointing decisions are made. We assume that a sub-plan brings together a set of related decisions that affect each other, but are independent of the decisions made in other sub-plans. In other words, the goal is to break the large plan space S_G into independent subspaces S_G^i such that $S_G = \bigcup S_G^{(i)}$. Within each $G^{(i)}$, we take brute-force approach to get an optimal solution among all possibilities.

While the plan dividing is arbitrary, we generate the sub-plans based on a key insight: how checkpointing decisions affect each other. In theory, a decision to making a checkpoint for a specific operator can influence the choice of a checkpoint of any other operator. But from our experience, the checkpointing decision for an operator mainly affects the decision for its successors. For example, a typical case is that when an operator A produces large data and its successor B produces small data with light computations, then it is better to checkpoint B rather than A. Based on this observation, we assume that checkpointing decision for an operator is independent from that of operators who are not its successors.

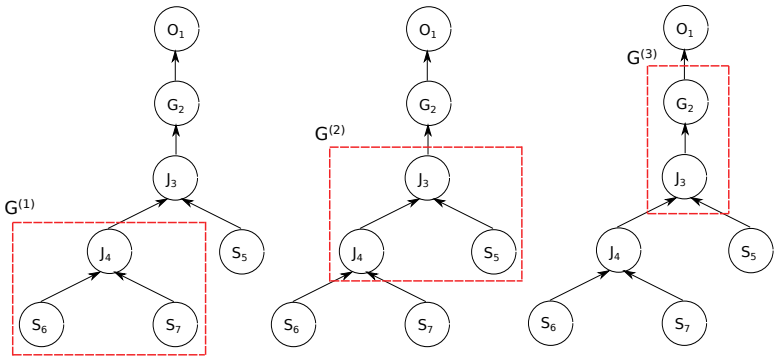

Figure 6.3: An example for the sub-plan generation

Sub-plan Generation: The sub-plans are generated based on the assumption mentioned above: the checkpointing decision for an operator does not affect the decision for operators who are not its successors. More specifically, when two operators O_i and O_k are separated by one or more operators in the query plan, the

checkpointing decision for O_i can be made independently from the decision made for O_k. Each sub-plan consists of one or several sibling operators and their unique successor. Sibling operators with a same successor are executed concurrently and the checkpointing decisions for them are influenced by each other as shown in the algorithm *Sibling Checkpointing*.

We take a query plan for TPC-H Query 3 as an example (Fig 6.3). As the algorithm traverses the plan in topological order, the first sub-plan $G^{(1)}$ consists of operator J_4 and its two predecessors S_6 and S_7. The next sub-plan $G^{(2)}$ comes with J_4, its sibling S_5 and its successor J_3. When an operator has no siblings, the sub-plan only contains two operators such as $G^{(3)}$.

Search in a Sub-plan: For each sub-plan $G^{(i)}$, the algorithm is to make the optimal checkpointing decisions for all operators in the sub-plan. As checkpointing an operator mainly brings benefit to the recovery of its successor, the algorithm adds a virtual operator O_0 to the top of the sub-plan and minimizes the expected completion time of O_0 calculated by the equation 6.10. The related variables for O_0 can be randomly specified because they don't affected the decisions for other operators. To get the optimal solution that minimizes the expected completion time of O_0, we use a brute-force technique to enumerate through that subspace.

The number of operators within any individual sub-plan is typically small. With a query plan presented as a binary tree with n operators, the number of operators in each sub-plan is at most 3 and the number of sub-plans is between $\frac{n+1}{2} - 1$ (for *full binary tree* in which every operator other than the leaves has two children) and $n - 1$ (for *degenerate binary tree* in which every operator only has one child) where $n > 1$. So the search space is varying from $(\frac{n+1}{2} - 1) \times 2^3$ to $(n - 1) \times 2^3$ which is much smaller than the original 2^n.

Algorithm: Overall, the selective checkpointing algorithm is described as follows: (1) For a given query plan G, the algorithm firstly finds the candidate operators to be checkpointed based on the *Sibling Checkpointing Algorithm* described above; (2) Then it generates the first sub-plan; (3) It enumerates all solutions within the sub-plan to find the optimal solution for the sub-plan; (4) It generates the next sub-plan and repeats the process until the entire plan is visited.

Algorithm 3: Algorithm for selecting the operators who are worth being checkpointed

Input: G: a query plan

Output: G with decided checkpoint values for each operator

find candidate checkpointing operators based on the Sibling Checkpointing and single Checkpointing algorithm ;

$G_{cur} \leftarrow first\_sub\_plan$;

while G_{cur} *is not None* **do**

  $find\ optimal\ solution\ for\ G_{cur}$ *based on equation* 6.12;

  calculate the intersection of the shape A and the ray OE:F;

  $G_{cur} \leftarrow next\_sub\_plan$;

Chapter 7

Evaluation

Contents

7.1 Experimental Environment

We conducted all experiments in a 32-node cluster. Each node uses 2.40 GHz Intel Xeon processor with 8 cores running 64-bit Debian 6.0 with 24GB RAM and 500G SATA hard disk. According to hdparm, the hard disks deliver 86MB/sec for buffered reads.

7.1.1 Compared Systems

We compare ParaLite with a commercial parallel database system DBMS-X from a major relational database company and a popular MapReduce system Hive.

DBMS-X: We installed the newest release of DBMS-X, a parallel row-oriented SQL DBMS. The official TPC-H benchmark conducted by the DBMS-X vendor used a slightly older version of the system. We specified our parameters for our installation the same with that in the official TPC-H benchmark. Specially, we did not enable the replication features in DBMS-X because all queries in the benchmark are read-only and enabling replication features makes it more complex for the installation process. We installed this version on each node. To enable the data partition feature, we have to install it as a root user which brings many troubles for us because we usually don't have the root authority.

Hive and Hadoop: For experiments in this paper, we used Hive version 0.8.1 and Hadoop version 1.0.3, running on Java 1.6.0. We configured both systems according to the suggestions offered by members of Hive ' s development team in their report on running TPC-H on Hive[58]. To reflect our hardware capacity, we configured the system to run eight Map instances and eight Reduce instances concurrently on each node. We also allowed JVM to be reused by all tasks instead of starting a new process for each Map/Reduce task. To make the comparison fair, we stored all input and output data in HDFS with the settings of one replica per block and without compression.

7.1.2 Dataset

The datasets used in all experiments are popular and widely used in database community or natural language processing applications.

- **TPC-H benchmark:** The TPC-H Benchmark[107] defined by the Transaction Processing Performance Council (TPC) is a popular one for comparing database vendors. It consists of a set of business oriented ad-hoc queries and concurrent data modifications. The queries simulate the real-business environment and cover almost all kinds of relation operations. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. The performance metric reported by TPC-H reflects multiple aspects of the capability of the system to process queries. These aspects include the selected database size against which the queries are executed, the query processing power when queries are submitted by a single stream, and the query throughput when queries are submitted by multiple concurrent users.

So far, many database vendors have published their results on performing TPC-H benchmark, including IBM DB2[29], Oracle Database[81], Microsoft SQL Server[92] and Sybase[98].

- **Medline:** MEDLINE (Medical Literature Analysis and Retrieval System Online)[41, 76] is the most authoritative international biomedical literature database of life sciences and biomedical information. It includes bibliographic information for articles from academic journals covering medicine, nursing, pharmacy, dentistry, veterinary medicine, and health care. MEDLINE also covers much of the literature in biology and biochemistry, as well as fields such as molecular evolution. MEDLINE contains more than 21.6 million records from 5,582 selected publications covering biomedicine and health from 1950 to the present. It is produced by the United States National Library of Medicine (NLM) and freely available on the Internet and searchable via PubMed[86]. MEDLINE is widely used in Biomedical Natural Language Processing (BioNLP) applications.

  Each article is stored in XML format in MEDLINE. The description of the article is defined as separated tags such as title, author, year and abstract. In all experiments of this section, by "the MEDLINE data", we actually mean the abstracts of many articles extracted from the original XML files.

7.2 Data Preparing and Loading

The TPC-H benchmark data were generated in parallel on every node using the dbgen program provided by TPC. We used the appropriate parameters to produce a consistent dataset across the cluster. As the data preparing and loading process for both TPC-H and MEDLINE data set are almost the same, here we only report the results of loading TPC-H data set. The time for loading 100GB data in total (with scaling factor of 100) to 16 nodes for each system is shown in Table. 7.1.

DBMS-X: We followed the suggestions from DBMS-X vendor to create the tables and indices, and to distribute data across the cluster. All tables were hash-partitioned across the nodes by their primary keys while *PartSupp* and *LineItem* relations were hash-partitioned on only the first column of their primary keys. In addition to creating index on the primary key for each table, the *Supplier* and *Customer* relations were indexed on their nation keys respectively, and the *Nation* table was indexed on its region column. Finally, the *LineItem* and *Orders* relations were organized by the month of the date columns for a partial ordering by date on each node of the cluster. The loading

DBMS-X(DL)	DBMS-X(IC)	ParaLite(DL)	ParaLite(IC)	Hive
11460	598	7980	695	420

Table 7.1: Data Preparing Time (seconds): DL–Data Loading IC–Index Creating

process worked as follows: data were first partitioned across the cluster and then the partitioned data were loaded on each node in bulk. DBMS-X took 3 hours and 11 minutes to load all related data into the database. Besides, the index creation took about 10 minutes.

Hive and Hadoop: We first loaded the source data into HDFS using the Hadoop command-line utility. The utility was run in parallel on all nodes and copied unaltered data files into HDFS under a separate directory for each table. Each file was automatically broken into 128MB blocks and stored on a local DataNode. Then we executed Hive DDL scripts provided by the Hive development team special for TPC-H benchmark to put relational mapping on the files. Since the metadata creation cost is negligible, the entire data preparing time is considered as loading data into HDFS and it took only 7 minutes.

ParaLite: In ParaLite, all tables were hash-partitioned across the cluster and indexed on the same key with that in DBMS-X respectively. But ParaLite cannot organize the *LineItem* and *Orders* relations by the month of their date columns. The main process of loading data is almost the same with DBMS-X. It first parses each record and sends it to the correct partition. Then each node loads received records to the SQLite database locally in parallel. The whole process took about 2 hours and 13 minutes and the index creating took about 11 minutes.

7.3 Evaluation of General Query Performance

In this section, we evaluated the scalability of ParaLite and the performance of ParaLite for both general queries and executables compared to Hive and DBMS-X.

7.3.1 Scalability of Typical SQL Task

We firstly performed three typical SQL tasks: selection, aggregation and join.

Selection Task:

The query shown below performs a lightweight filter to find the related information in the *Orders* table with the *orderkey* smaller than a user-defined threshold. In

this experiment, we set this threshold parameter to 10,000,000, which yields approximately 250,000 records per node. ParaLite directly dispatches the query to each node and executes it by SQLite in parallel. Hive uses only a single Map function that parses *Orders* tuples, and outputs the records as new key/value pairs if the *orderkey* predicate succeeds. This query does not require a Reduce function.

```
SELECT orderkey, custkey, orderstatus, totalprice
FROM   Orders
WHERE  orderkey < 1000000
```

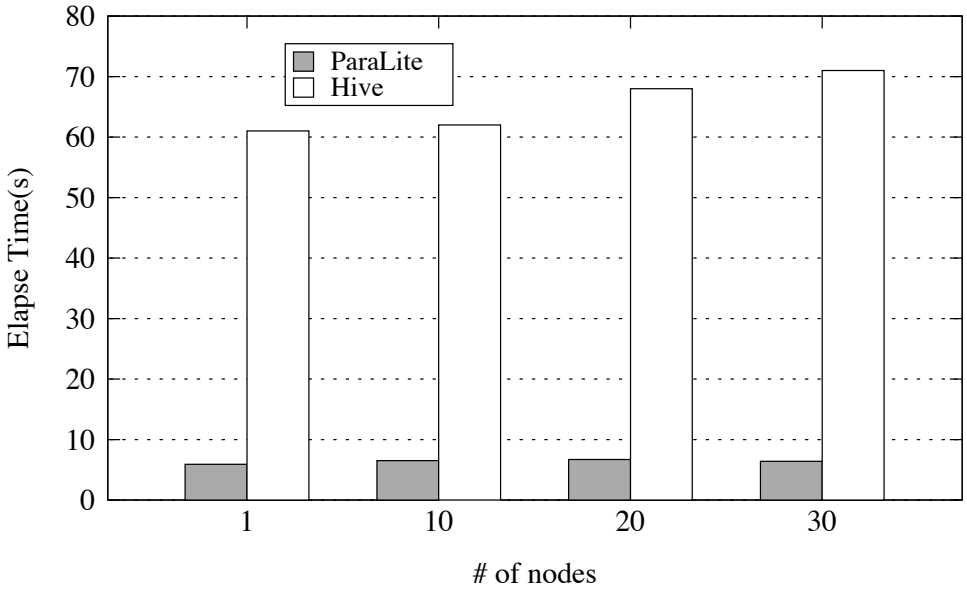

Figure 7.1: Scalability for Performing Selection Query

The number of data nodes varies from 1 to 30 with about 2GB data on each node. From Fig. 7.1, we can see that both ParaLite and Hive scale well with the increase of data nodes. The overall execution time of Hive slightly increases mainly because that the increase of result tuples leads to longer time for storing them. Moreover, the experimental results show that ParaLite is about 10 times faster than Hive. On the one hand, ParaLite creates index on column *orderkey* for table *Orders* which reduces the data access time significantly. On the other hand, Hive performs sequential scan of all tuples to parse each tuple and check if the value of *orderkey* satisfies the threshold. In addition, Hadoop ' s start-up cost should not be ignored. For a cluster with 30 nodes, it takes about 10 to 15 seconds from the submission of a job to the execution of the first Map task.

Aggregation Task:

The aggregation query performed by ParaLite and Hive is described below. As the table *LineItem* is not partitioned on either column *returnflag* or *linestatus*, ParaLite transformed the query to an execution plan with two operators. A *sub-query* operator which performs a local aggregation, hash-partitions the result tuples on the group key and distributes them to the followed *group* operator to perform a global aggregation. Hive uses one MapReduce job to finish the query which consists of both a Map and Reduce function. The Map function outputs all tuples from *LineItem* which are hash-partitioned on the group key and sent to the Reduce function that aggregates the sum of each of *extendedprice* and other related columns.

```
SELECT   returnflag, linestatus, sum(quantity), count(*),
         sum(extendedprice), sum(extendedprice * (1 - discount))
FROM     LineItem
GROUB BY returnflag, linestatus
```

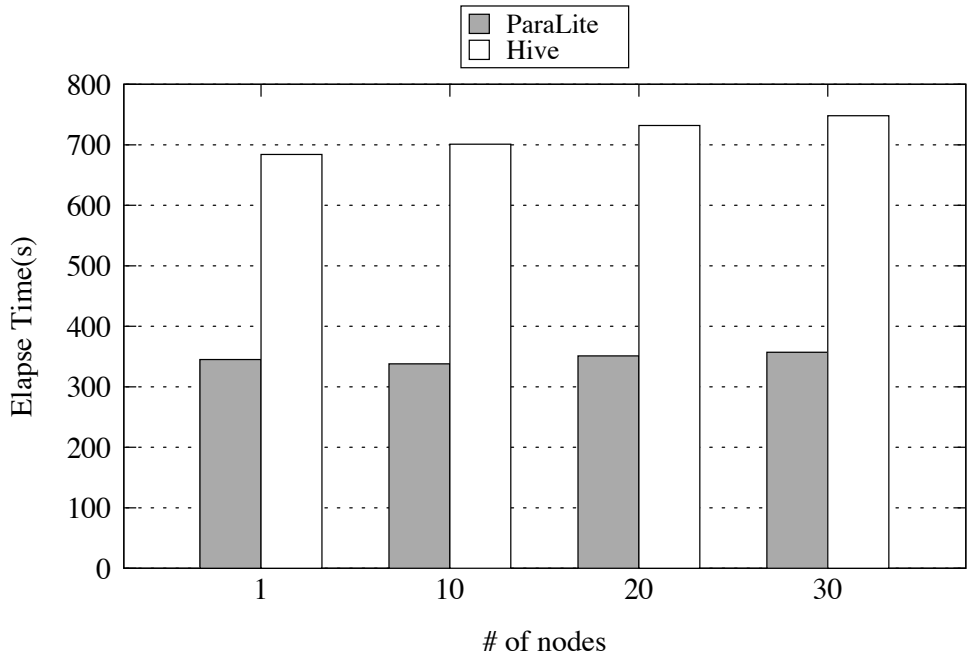

Figure 7.2: Scalability for Performing Aggregation Query

The number of data nodes varies from 1 to 30 with about 10 GB on each node. Fig. 7.2 firstly shows that ParaLite scales out well across the cluster. The execution time of the query increases little with the increase of data nodes. As mentioned above, ParaLite firstly aggregates data locally by SQLite on each node and then

distributes data to the group operator to perform the final global aggregation. The local aggregation produces only 4 groups, so there is little data transfer between the two operators and the overhead for the global aggregation can be ignored as the input data is really small. Therefore, the execution doesn't not suffer from the bottleneck of data transfer and the overall execution time is decided by the performance of SQLite. While ParaLite reduces the data transfer overhead by taking advantage of local aggregation, Hive has to retrieve, materialize and distribute all tuples to the Reducer. As a result, ParaLite is about twice faster than Hive.

Join Task:

We perform two join queries:

- Join Query 1 (J1): The query described below is to get the order information of a product by joining tables *LineItem* and *Orders*. As these two tables are hash-partitioned on the join key, ParaLite directly dispatches the query to each node to be executed by SQLite and merges the results. Hive uses a single MapReduce job with a Map function which filters all tuples whose order dates are over the predicate and a Reduce function which joins the satisfied tuples on the order key of both tables. Each node produces about 13M result data.

```
SELECT L.orderkey, O.orderdate, O.shippriority
FROM   Orders O, LineItem L
WHERE  L.orderkey = O.orderkey
       AND O.orderdate < '1995-10-11'
       AND L.shipdate > '1995-10-11'
```

- Join Query 2 (J2): The query described below gets the price information of products by joining tables *LineItem* and *Part*. As the two tables are not partitioned on the join key, the execution plan for the query in ParaLite consists a join operator with two children of sub-query operators which access tuples from each table with necessary predicates. Hive executes the query in the similar way used to perform the query J1. Each node produces about 340M result data.

```
SELECT L.extendedprice, L.discount
FROM   LineItem L, Part P
WHERE  L.partkey = P.partkey and L.shipdate >= '1995-10-11'
```

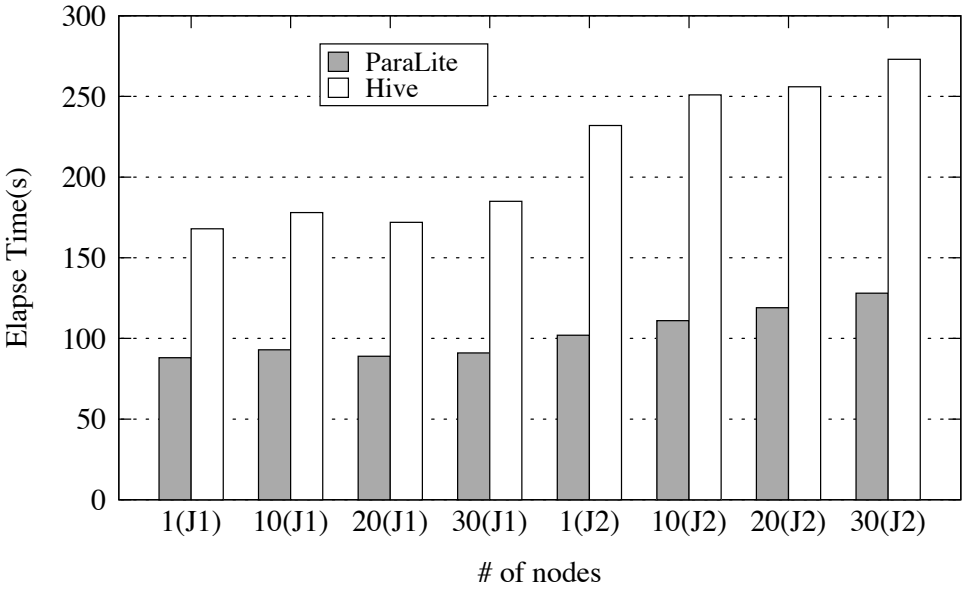

Figure 7.3: Scalability for Performing Join Query

As always, the number of data nodes increases from 1 to 30 with fixed size of data on each node (about 7.5GB *LineItem*, 1.7GB *Orders* and 220MB *Part*). From Fig. 7.3, ParaLite and Hive scale well for both queries. The reason for the slight increase of the overall execution time for J2 is the data to be transferred to the join operator (Reducer in Hive) has increased (330MB per node) and the result tuples to be stored in files are also increased from 340MB on a single node to about 10GB on 30 nodes. In addition, ParaLite is approximately twice faster than Hadoop for both queries. For the query J1, the main execution is performed by SQLite and the performance of SQLite is proved to be better than Hive. For the query J2, ParaLite executes the join operations using hash-join which is about 40% faster than sort-join used in Hive.

7.3.2 Scalability of Complex Task

Next, we perform the TPC-H Query 3, a more complex task with composition of selection, join, aggregation and sort. The execution plans of the query expressed by ParaLite and Hive respectively are shown in Fig. 7.4. As the two relations *Orders* and *LineItem* are partitioned on the join key, the join operation on them is pushed into a SQL query to be executed by SQLite directly. On the other hand, Hive needs another single MapReduce job to join these two relations.

In the experiments, we increased the nodes from 10 to 30 with about 10GB on each node. The results are shown in Fig. 7.5 as we expected: (1) Both systems

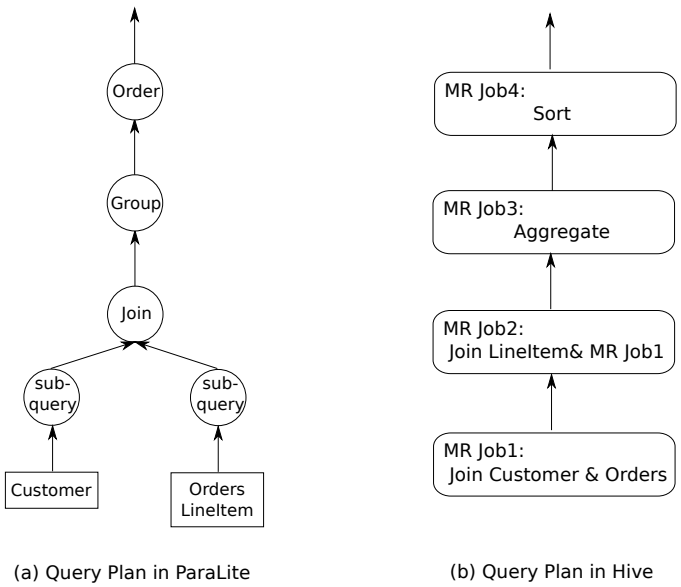

Figure 7.4: The Query Plan of TPC-H Query 3

scale well with the increase of data nodes; (2) ParaLite is about 4 times faster than Hive. As we explain before, firstly, join and aggregation are faster in ParaLite due to data partitioning. Then writing all intermediate data to durable storage in Hive degrades the overall performance. In addition, the start-up overhead of Hadoop cannot be ignored since there are 4 MapReduce jobs in total and each one takes about 15 seconds until the first Map task begins.

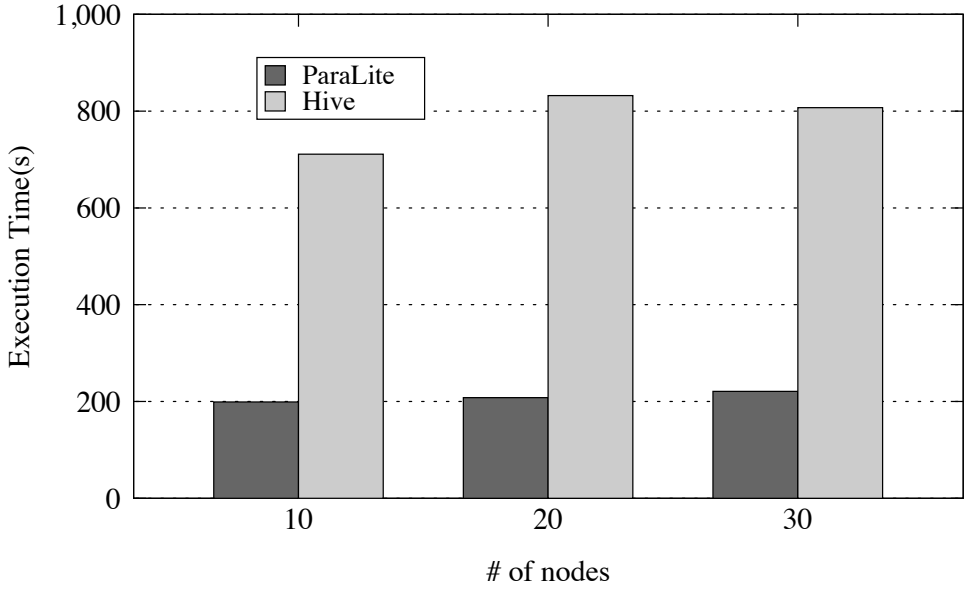

Figure 7.5: The performance of TPC-H Query 3

7.3.3 Comparison of Completion Time for TPC-H Query

We run TPC-H queries from Query 1 to 20 with scaling factor 100 in a cluster of 16 nodes. For DBMS-X and Hive we executed the queries as suggested in the official TPC-H reports by the vendors. Since the syntax of HiveQL is just a subset of SQL, for may queries, the original TPC-H queries were rewritten into a series of simple queries which produce the desired results in the last step in Hive. For ParaLite, several queries are also rewritten into a series of simple queries as Hive does and Query 7, 11 cannot run successfully because currently ParaLite does not support operations like *left join* and nested query in *where* clause.

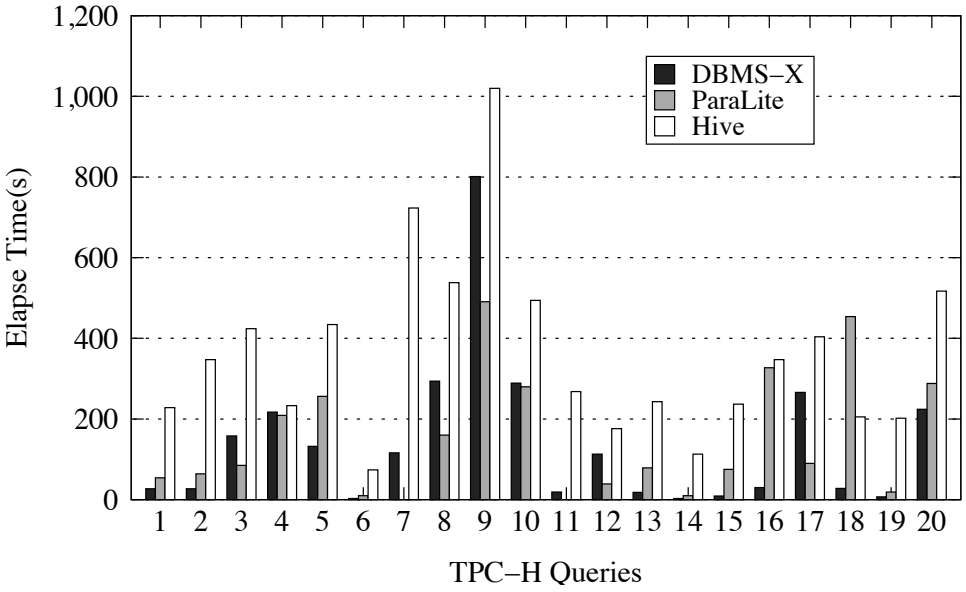

Figure 7.6: TPC-H Performance of Several Approaches

Fig. 7.6 shows the benchmarking results for all three systems. First, it is not surprising that DBMS-X and ParaLite significantly outperform Hive for almost all queries. These results are expected because the Hive development team reported a similar result when comparing Hive with DBMS-X[58]. The main reason for the superior performance of DBMS-X ' s and ParaLite is the ability to take advantages of partitioning and indexing. Without this ability, Hive performs a full data scan for every selection and most of the joins in Hive require to repartition and shuffle all records across the cluster. However, ParaLite is slower than Hive for Query 18. The reason for the inferior performance is the intermediate data storing to database. ParaLite and Hive rewrite this query into several simpler ones. ParaLite stores the output of each query to the database and it takes much time than storing them to HDFS in Hive.

Secondly, the performance of ParaLite and DBMS-X are comparable. ParaLite loses for some queries and slightly wins for other queries. The main reason for the inferior performance is the lack of organization of data. DBMS-X organizes the *LineItem* and *Orders* relations by the month of their date columns for a partial ordering by date. DBMS-X gains much from the organization of data for many queries such as Query 2, 15. Another reason is that for some rewritten queries, ParaLite is required to store intermediate data from each step to database as mentioned above while DBMS-X does not need to materialize any of them.

Obviously, the biggest bottleneck in ParaLite is the storing of intermediate data from each simple query. This is limited by the types of queries supported by ParaLite. In the future, we plan to refine ParaLite to support the full syntax of SQL. However, the experimental results still show the efficiency of ParaLite for most queries.

7.4 Evaluation of Executable Performance

Providing the straightforward and efficient integration of external executables into query plans is a major feature of ParaLite for workflows. In this section, we run several queries with the integration of both heavy and lightweight executables.

Before we show the experimental results, we briefly introduce the executables used in our experiments and how to express them in the three systems.

Heavy (CPU-Intensive) Executable: Enju

Enju[39] is a fast, accurate, and deep parser for English text and widely used in natural language processing (NLP) applications. It reads data from the standard input and writes the result to standard output. The parse results for each sentence contains much information and are presented in multiple lines. The results are separated by an empty line for different sentences. *Enju* is a cpu-intensive program with very high start-up overhead. It firstly needs to load dictionary before processing sentences which takes about 8 seconds.

ParaLite expresses the task simply by a query with a UDX definition as follows. The UDX *F* takes a column *sentence* as the argument and returns a new column *enju_result*. By setting the *output_row_delimiter* a empty line, the whole query returns records with two columns and each of record is the sentence ID and the parsed results of the sentence by enju.

```
SELECT sid, F(sentence) as enju_result
FROM   Abstract
with   F = "Enju" output_row_delimiter EMPTY_LINE
```

DBMS-X performs the task with an UDF in the query below. However, the function F is developed conforming to the database API. In the experiments, we implement a java program which receives each sentence, starts the Enju process and returns the result tuple. Besides, before the query is issued, we have to register the definition of the function into the database system. The query returns the same records with ParaLite.

```
SELECT sid, F(sentence) as enju_result
FROM   Abstract
```

Hive provides the syntax to integrate any executable into the HiveQL straightforwardly. It runs the executable as a MapReduce job. However, it cannot map the sentence ID to the enju result for the sentence. So we encapsulate the executable in another program to fulfill the mapping. The program reads records with two columns, feds only sentences to the enju program and maps the output of enju of a sentence to its ID.

```
from (
      from Abstract map sid, sentence
                   using 'enju_wrap' as SID, enju
     ) map_output
select map_output.sid, map_output.enju
```

Lightweight Executable: simple_tokenizer.pl

The executable *simple_ tokenizer.pl* is also a NLP tool to tokenize the words in sentences. It reads sentences from standard input and returns the sentences with tokenzied words. Similar with *Enju*, this executable is expressed by ParaLite, DBMS-X and Hive in the same way. For example, ParaLite performs it using the following query:

```
SELECT sid, S(sentence) as tk_sentence
FROM   Abstract
with   S = "perl simple_tokenizer.pl"
```

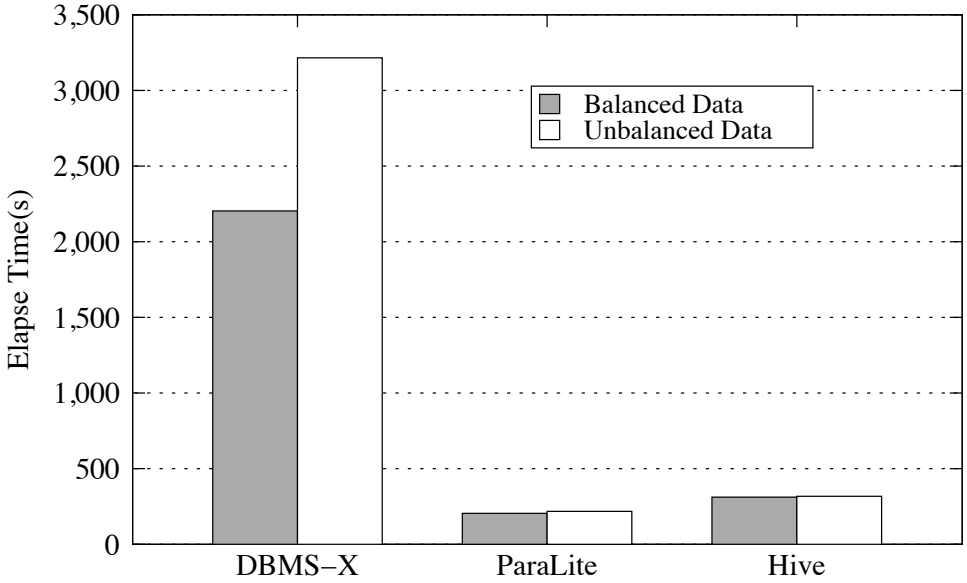

Figure 7.7: Completion Time of the Heavy Executable

7.4.1 Comparison of Completion Time

We tested the completion time of the two executables with both balanced and unbalanced data across a 16-node cluster.

For the heavy executable, 1.6MB data are distributed across the cluster. Each node has 0.1MB data when the data are evenly distributed while each of 8 nodes holds 0.15MB and each of the other nodes has only 0.05MB data when data are not evenly distributed. The results are shown in Fig. 7.7. Firstly, ParaLite is slightly faster than Hive but about 10 times faster than DBMS-X in both situations. The main reason for the inferior performance of DBMS-X is the high start-up overhead of enju. DBMS-X takes about 8 seconds to initiate a enju process for each sentence while ParaLite and Hive take the 8 seconds for a bulk of data (50KB in our experiments). Secondly, ParaLite has similar performance no matter whether data are evenly distributed or not, so does Hive. However, there exists big differences between the completion time of both cases for DBMS-X because it uses static data scheduling policy for the parallelization of executables. Once data are partitioned, DBMS-X assigns a processor for a partition of data. When the assigned data is finished, the processor does not get the data from another partition. ParaLite and Hive take dynamic data scheduling policies in which once a processor becomes idle, data from other partitions are dispatched to it.

For the lightweight executable, 32GB data are distributed across the cluster. Each node has 2GB data when the data are evenly distributed while each of 8

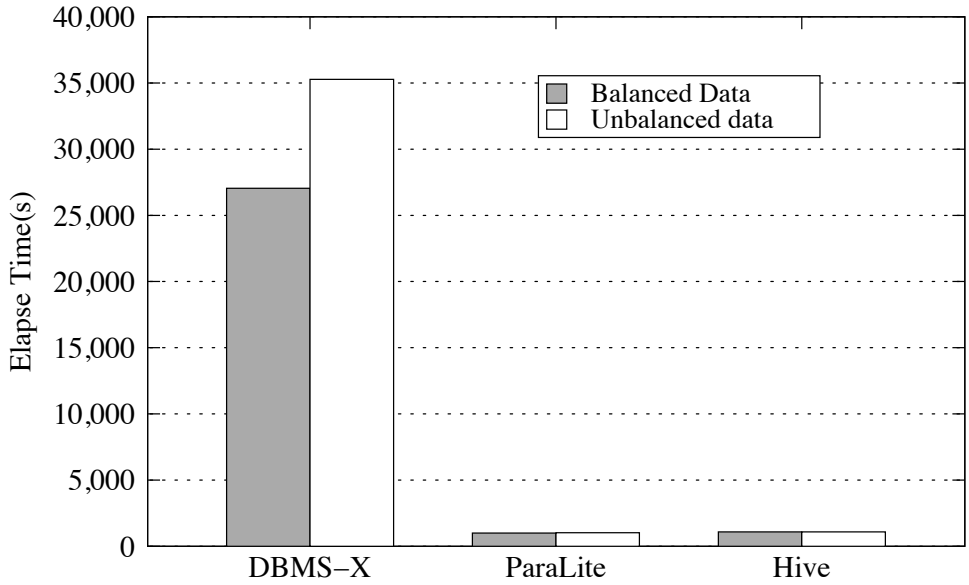

Figure 7.8: Completion Time of the Lightweight Executable

nodes holds 3GB and each of the other nodes has only 1GB data when data are not evenly distributed. Fig. 7.8 shows the completion time with the two types of data distribution. The results are similar with that for the heavy executable. ParaLite and Hive are 25 times faster than DBMS-X. Even if *simple_ tokenizer.pl* does not have high start-up overhead, the time for a large number of processes creation is considerable. Similar with the heavy executable execution, DBMS-X has worse performance when data are not evenly distributed due to the lack of efficient data scheduling policy.

7.4.2 Scalability of Executable

We test the scalability of ParaLite when the executable is performed with the increase of computing clients. The data distribution for all executable related experiments in the rest of this section is set as follows. For the heavy executable, 30MB data are distributed across a 30-node cluster. Each node has 1MB data when the data are evenly distributed while each of 15 nodes holds 2MB data when data are not evenly distributed. For the lightweight executable, 120GB data are distributed across the same cluster. Each node has 4GB data when the data are evenly distributed while each of 15 nodes holds 8GB data when data are not evenly distributed. The number of clients varied from 8 to 240 for both executables.

As Fig. 7.9 and Fig. 7.10 show, the speedup is close to the ideal (linear) one. The reasons for the small deviation are: (1) the time for accessing data from the database is not decreased with the increase of clients; (2) when data are unevenly

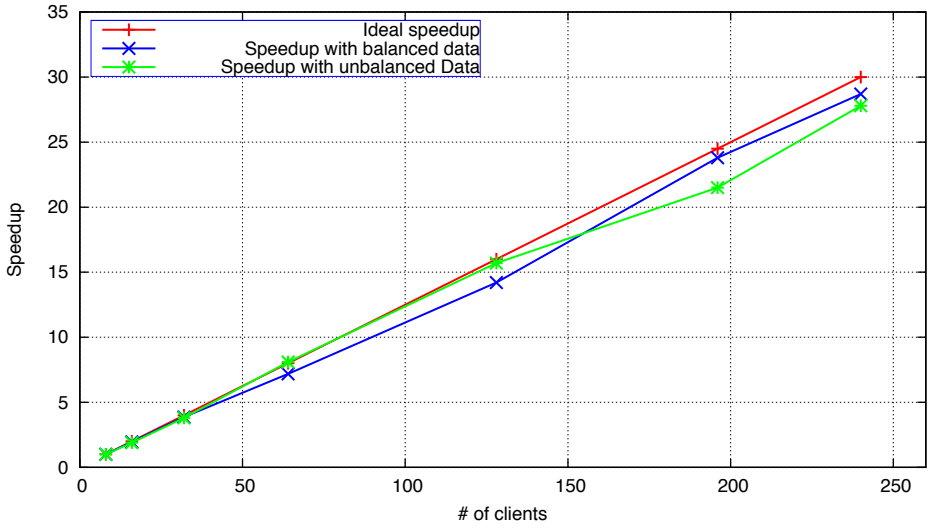

Figure 7.9: Speedup for the Heavy Executable

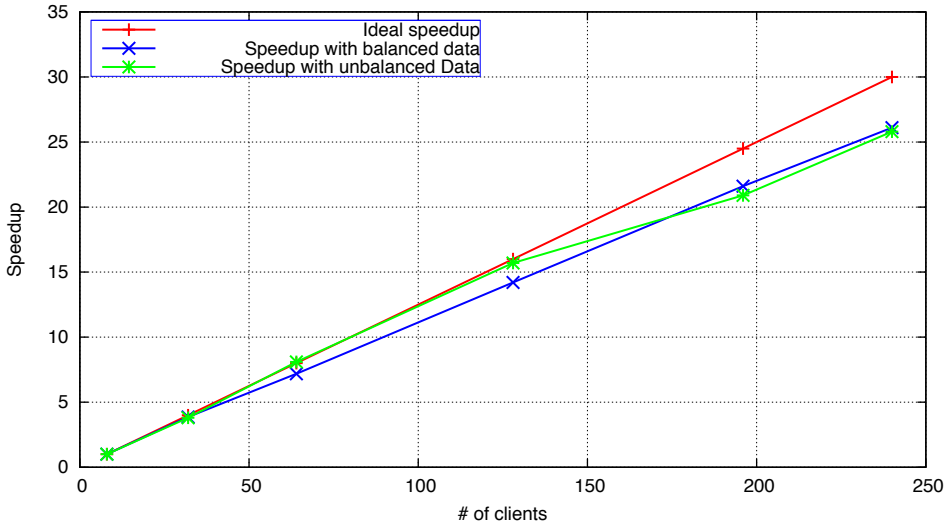

Figure 7.10: Speedup for the Lightweight Executable

distributed, data transfer is increased when the clients are located in the nodes who don't have data.

7.4.3 Impact of Block Size

The parallelization of executables in ParaLite is affected by the size of block. A block of data is scheduled at a time. In this set of experiments, data are evenly distributed across the cluster and the number of clients is 128. The results are shown in Fig. 7.11 and Fig. 7.12 respectively for the heavy and lightweight executables.

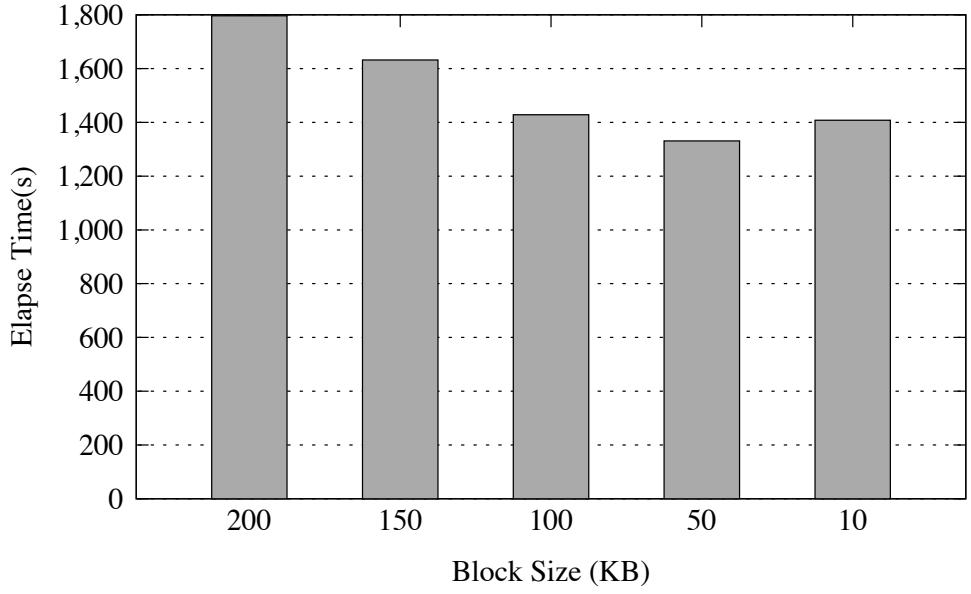

Figure 7.11: Impact of Block Size for the Heavy Executable

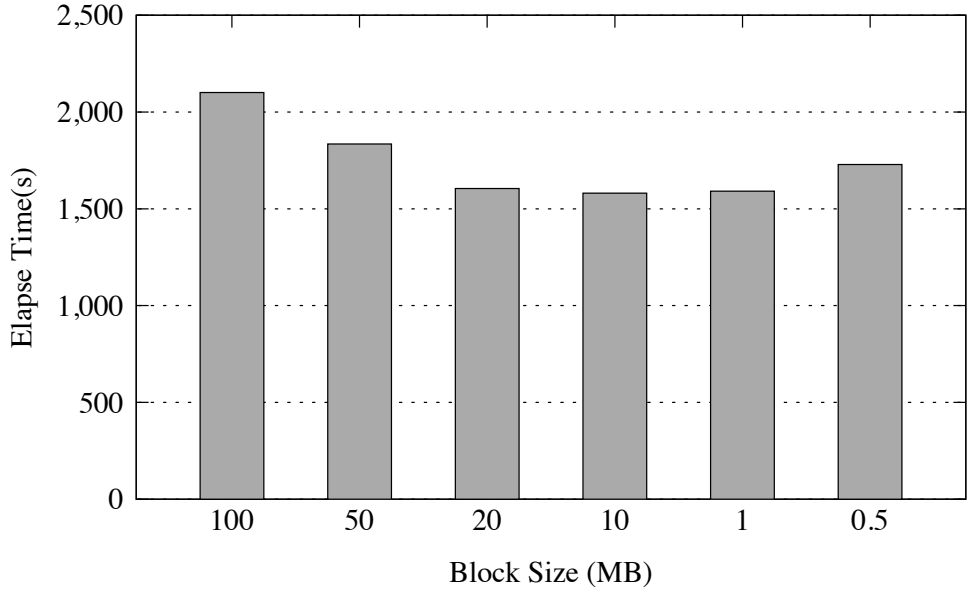

Figure 7.12: Impact of Block Size for the Lightweight Executable

From the results, we can see that the completion time decreases when the size of block is reduced. However, when the block size is reduced to a certain extent, the completion time starts to increase. The reason for the decrease of the completion time is coarse-grained parallelization. With large block size, it is difficult to rigorously balance the load of all clients. Although ParaLite achieves the load

balancing by greedy assignment of task, it is still possible that one client gets the last task while all other clients just finish. In this worst situation, the processing of the last block of data becomes a potential bottleneck. For the heavy executable, the reason for the increase of the completion time is the start-up overhead of the executable. When the block size becomes very small, the ratio of the start-up overhead to the overall execution time becomes larger. For the lightweight executable, with too small block size, the overhead for scheduling, dispatching data and handling the information from clients increases. So we allow the user to specify the block size as they know better about their executables and data than ParaLite.

7.4.4 Evaluation of Data-distribution Algorithm

In this section, we evaluate our DLLB (Data Locality with taking Load Balance into consideration) algorithm for the data distribution in both normal and abnormal situations. We run 30 computing clients with one client on each node.

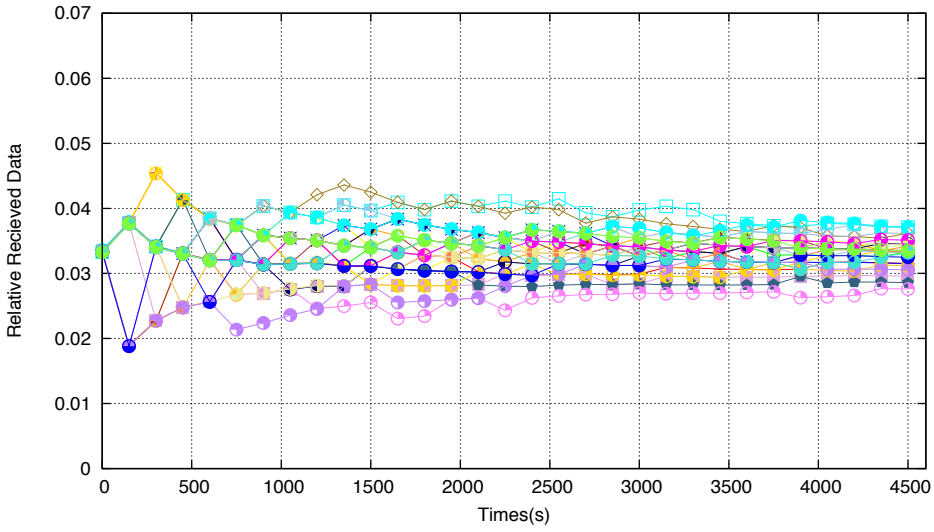

Figure 7.13: Load Balancing Test for Heavy Executable in Normal Situation

In the normal situation that no machine is artificially loaded, the load balance for the cluster over time for both executables are shown in Fig. 7.13 and Fig. 7.14. The Y-axis is the fraction of whole data that each client receives. If the system is perfectly load balanced, all clients should receive equal size of data. The system works exactly as we expected while performing the lightweight executable and the 30 clients hover around 0.033 of the whole load. A small deviation from our expectation exists for the execution of the heavy executable. From our observation, the abilities of all clients to perform the heavy executable are not perfectly equal.

Some clients are several seconds faster than others for a block of data, leading faster clients getting more data. However, the lines in Fig. 7.13 tending to be straight indicate that they get data at the same speed. Most of data are executed locally and only 463KB and 500MB data are transferred through network for heavy and lightweight executables respectively.

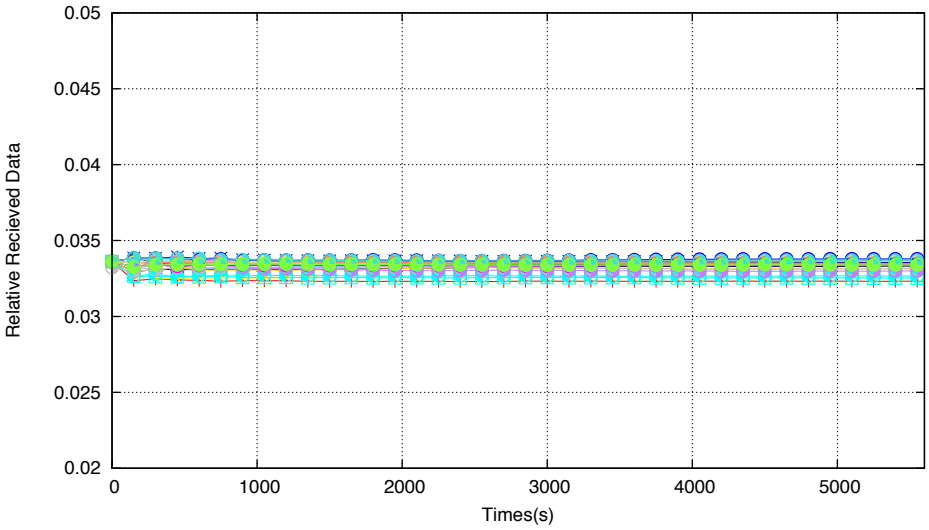

Figure 7.14: Load Balancing Test for Lightweight Executable in Normal Situation

In the abnormal situation, we adopted a system stressing utility called CPU Burn-in [27] to artificially load a given machine. CPU Burn-in spawns a number of processes to consume system resources. At $t = 800$, we start CPU Burn-in on *client18*. Fig. 7.15 and Fig. 7.16 show the results for the heavy and lightweight executables respectively. Before we push the stress on the client, the load is balanced as in the normal situation. After the client is artificially loaded, its received data decreases and other clients compensate for the over-loaded client's loss quickly. The data transferred through network is 1025KB for the heavy executable and 1.6GB for the lightweight one.

7.5 Evaluation of Data Model

To test the efficiency of `Interleaved Declustering`, we use TPC-H Query 6 which is directly executed by each SQLite instance and all results are simply merged and returned to the client. In this experiment, 10 data nodes are used and data set comes from TPC-H benchmark with the scaling factor 100. The size of relation `LineItem` is about 80GB.

```
select sum(extendedprice * discount) as revenue
```

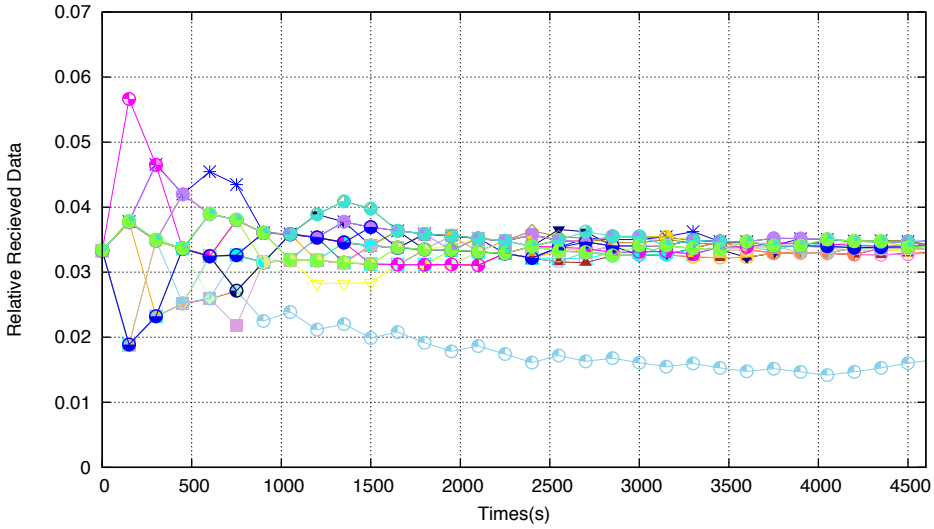

Figure 7.15: Load Balancing Test for Heavy Executable in Abnormal Situation

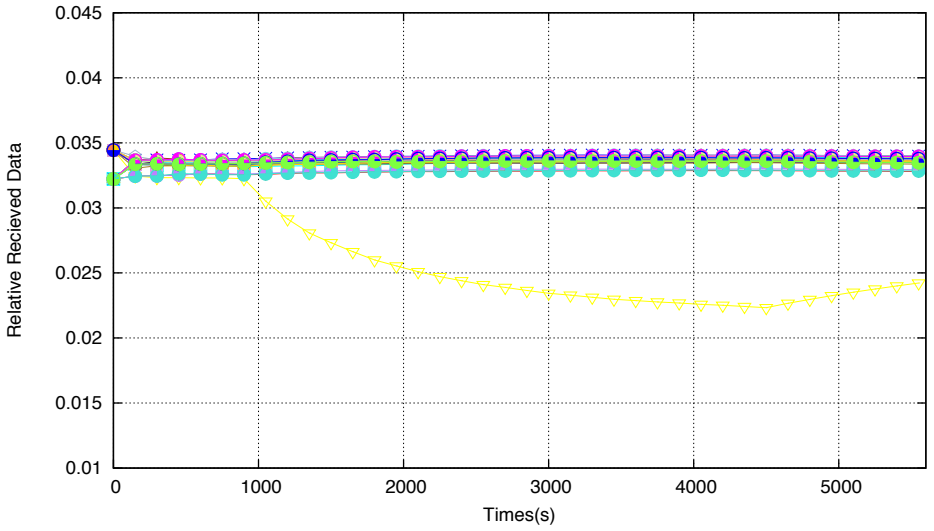

Figure 7.16: Load Balancing Test for Lightweight Executable in Abnormal Situation

```
from LineItem
where shipdate >= date('1995-10-11')
      and shipdate < date('1995-10-11', '+1 year')
      and discount between 0.1 - 0.01 and 0.1 + 0.01
      and quantity < 50
```

Firstly, data are partitioned across the 10 nodes by `Interleaved Declustering` and the sub-cluster size is 10. Each partition is divided into 45 chunks. In this case,

only one data node is allowed to be failed. Fig. 7.17 shows the load balance of the cluster over time. The y-axis is the relative ratio of the computation time for each node to the total computation time. We can see that the ten nodes almost have the same ratio (≈ 0.1) which means the system is well load balanced. When $t = 70$, we terminate node 2. The remaining 9 nodes share the workload of node 2 and the load is still balanced since the replica of chunks on node 2 is evenly replicated on the left 9 nodes.

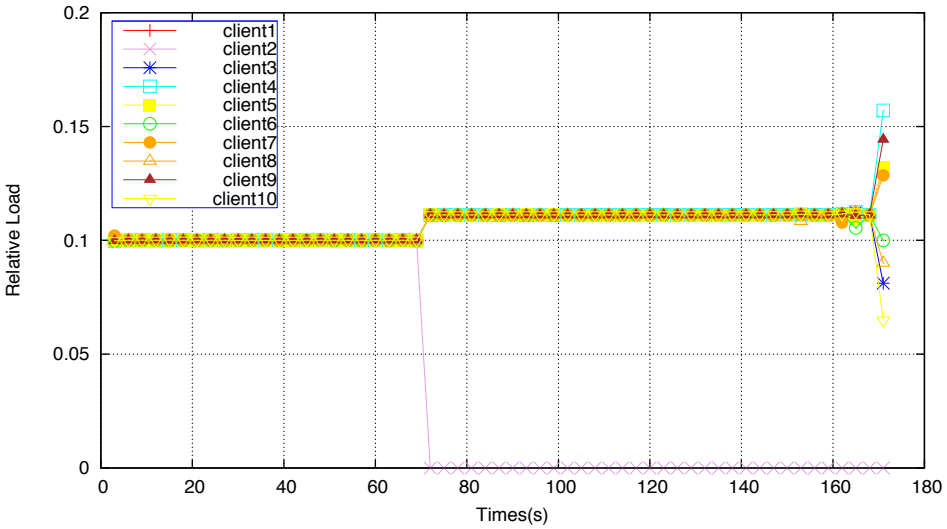

Figure 7.17: Load balancing with 1 node failure

Next, we set the sub-cluster size to be 5 and re-load data into the 10 data nodes. In this experiment, each node has 40 chunks which are replicated on other nodes in the same sub-cluster. From Fig. 7.18, we see that the load is well balanced in this setting. At time $t = 80$, we terminate two nodes (node 2 and node 8) in two different sub-cluster. The workload on each failed node is compensated by the other 4 nodes in the same sub-cluster.

7.6 Evaluation of Selective Checkpointing Mechanism

The experiments are performed to verify: (1) Different fault-tolerant strategies heavily affect the overall runtimes of queries; (2) Our selective checkpointing mechanism can choose reasonable operators to be checkpointed; (3) The mechanism outperforms other fault-tolerant strategies; (4) the divide-and-conquer algorithm has smaller overhead than brute-force approach while keeps similar effectiveness; (5) our mechanism can achieve similar slowdown with Hive (Hadoop) upon a failure; (6) the system (ParaLite) scales well with the mechanism.

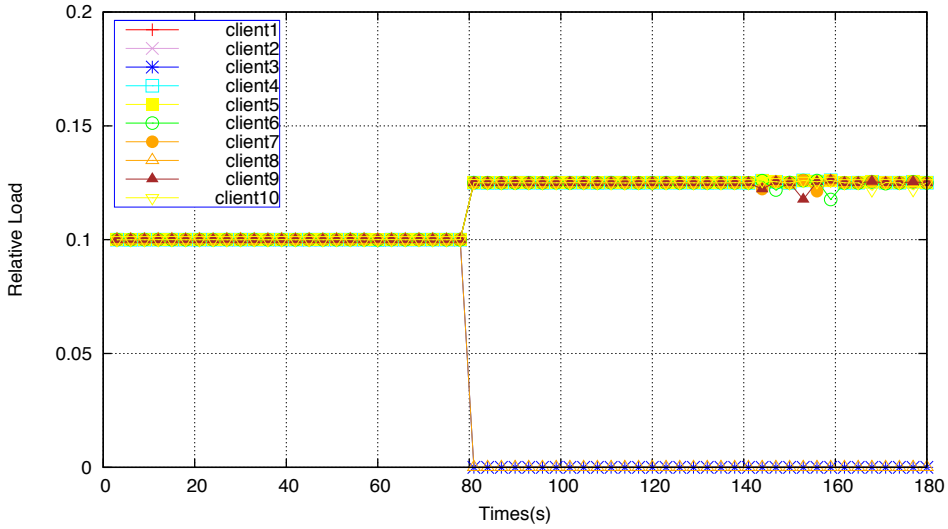

Figure 7.18: Load balancing with 2 node failures

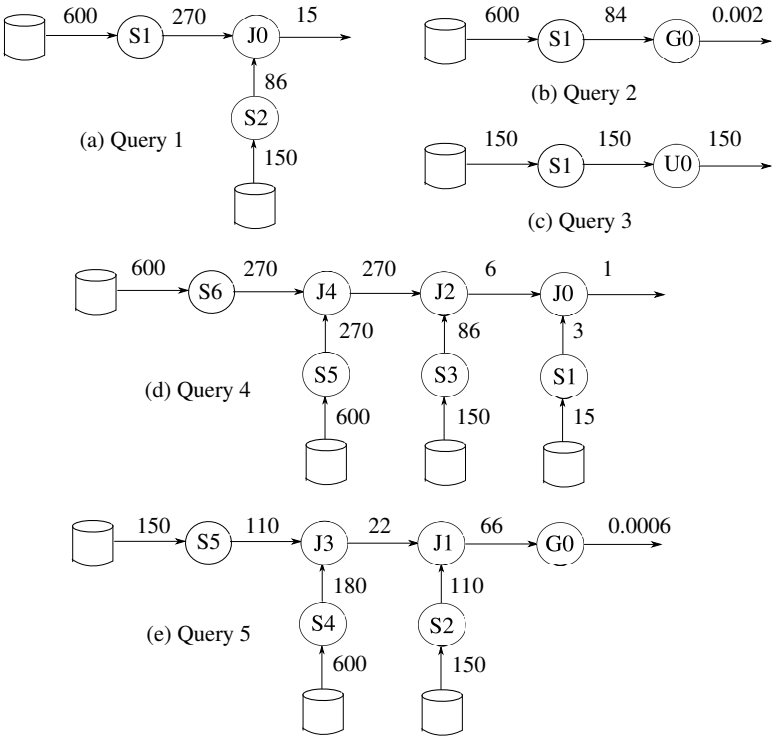

Figure 7.19: Query plans of several TPC-H queries: all numbers (tuples) are in millions

7.6.1 Effect of Fault-tolerant Strategy

We first evaluated that different fault-tolerant strategies affect the overall performance of a query plan. We performed the typical SQL tasks, including selection (S), join (J) and aggregation (G) through queries 1 and 2 from Fig. 7.19 and a special task with User-defined Executable (U) by query 3.

Recall that our cost model requires two functions for operator i: f_i provides the number of output tuples for a given number of input tuples and g_i gives the processing time for a given number of input tuples (shown in Eq. 6.5 and Eq. 6.6). Function f_i is determined by the selectivity of the operator. We define g_i based on our measurements. For example, for a sub-query, the execution time (seconds) is estimated as 3×10^{-6} times the number of input tuples.

Fig. 7.20 through Fig. 7.22 show the actual and predicted execution times for Queries 1 through 3. All X-axises are the fault-tolerant strategies for operators. For example, in Fig. 7.21, *NN* means No checkpoint for operator *S1* and *G0* while *NC* indicates No checkpoint for *S1* and a checkpoint for *G0*. Note that the order of the operators is from left to right in the query plans. For Query 1, to show the results clearly, we assume that no checkpoint for *S2*. We inject a failure at the middle of the completion.

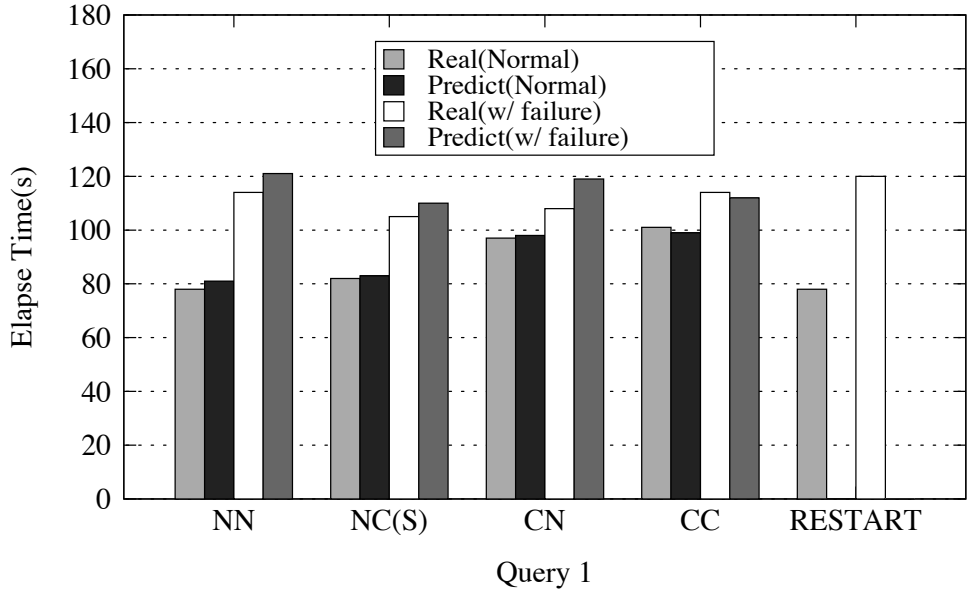

Figure 7.20: Elapse Query 1 (JOIN)

- Different fault-tolerant strategies heavily affect the overall runtimes of queries. The differences between the overall execution time with the best and worst

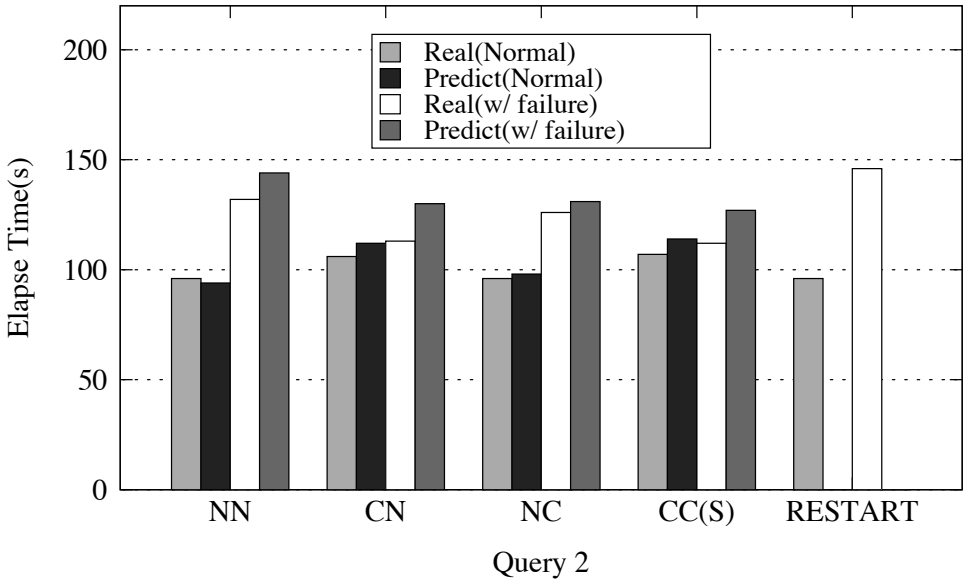

Figure 7.21: Query 2 (GROUP)

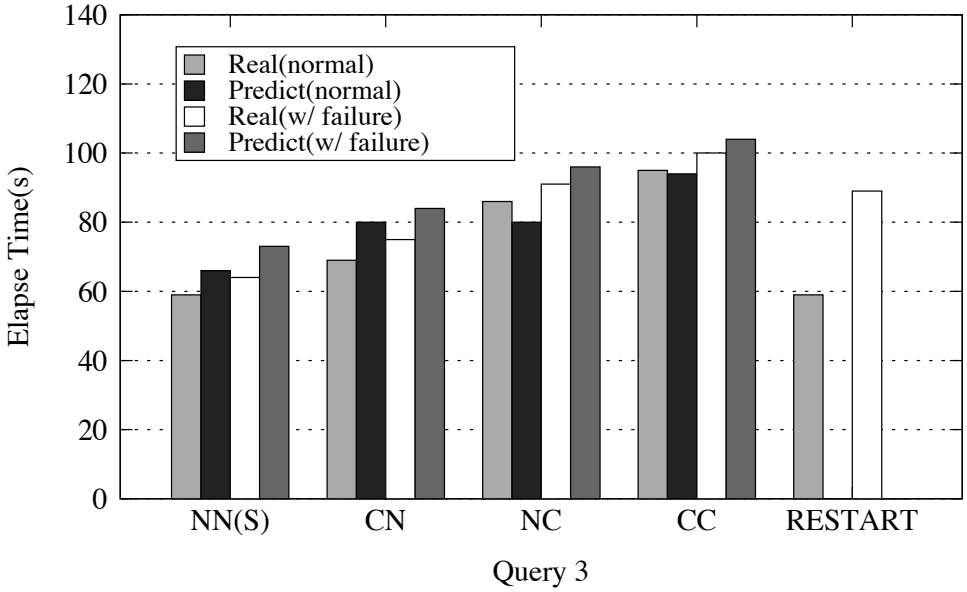

Figure 7.22: Query 3 (UDX)

strategy are high: For normal execution without a failure, the differences are 30% for Query 1, 15% for Query 2, and 61% for Query 3. For the execution recovered from a failure, the differences are 14% for Query 1, 30% for Query 2, and 56% for Query 3. Each of queries 1 through 3 achieves the best performance with a different fault-tolerant strategy in both failure-free (normal) and failure-prone environments. For all queries, checkpointing nothing (NN) is the

best option if no failure occurs. However, once a failure occurs, only Query 3 achieves the best performance with this strategy while Query 1 requires the strategy NC and Query 2 requires CC.

- Our selective checkpointing mechanism can identify the best strategy for all queries. The predicted execution time can not exactly match but close to the actual time. Most of the differences come from the simple model for the data transfer and the assumption that data are evenly distributed to each process which is not simply satisfied in real cases. However, the model could correctly get the relative order of predicted execution time of plans with different strategies. Our mechanism simulates the execution in a failure-prone environment and predicts the runtime of a query plan with the assumption that each operator has a fixed probability of failure, so it considers the strategy with the smallest predicted execution time with failures as the best one. Therefore, for Query 1, the option of only checkpointing the output of the join operator (NC) is chosen although the execution time with option NC in normal situation is slightly larger than that with the option of checkpointing nothing (NN). For Query 2, the strategy of checkpointing everything (CC) is considered as the best strategy by our mechanism. For Query 3, it decides to checkpointing nothing.

- Finally, restarting a query, the strategy used in most existing database systems, produces the largest slowdown upon a failure among all strategies. Strategies other than RESTART reduce recovery times with minimal impact on the execution time without failures. For Query 1 through 3, RESTART is 14%, 30% and 39% worse than the best strategy respectively.

7.6.2 Effectiveness of Selective Checkpointing

In this section, we apply several fault-tolerant strategies to two TPC-H queries, Query 4 and 5 from Fig. 7.19.

We compare the overhead for choosing checkpointing operators and the places of checkpointing with both *Divide-and-Conquer (DaC)* and *Brute-Force (BF)* algorithms. From the Table. 7.2, we can see that although the overhead with both algorithms is very small but *DaC* is several times faster than *BF*. The difference increases exponentially with the increase of operators. For the tested queries, the places for checkpointing are the same for both algorithm.

We inject a failure at 80% execution of Query 4 and 50% execution of Query 5. Fig. 7.23 and Fig. 7.24 shows the actual and predicted execution time with/without a failure. In the figures, CKNONE means that no operator is checkpointed; CKALL

	Overhead(second)		Checkpoints	
	Query4	Query5	Query4	Query5
Divide-and-Conquer	0.011	0.0095	S1, J2	G0,S2,J3,S4,S5
Brute-force	0.15	0.092	S1, J2	G0,S2,J3,S4,S5

Table 7.2: Divide-and-Conquer Compared with Brute-force

indicates all intermediate data are materialized; SELECTIVE shows the operators chosen by our selective checkpointing mechanism are checkpointed.

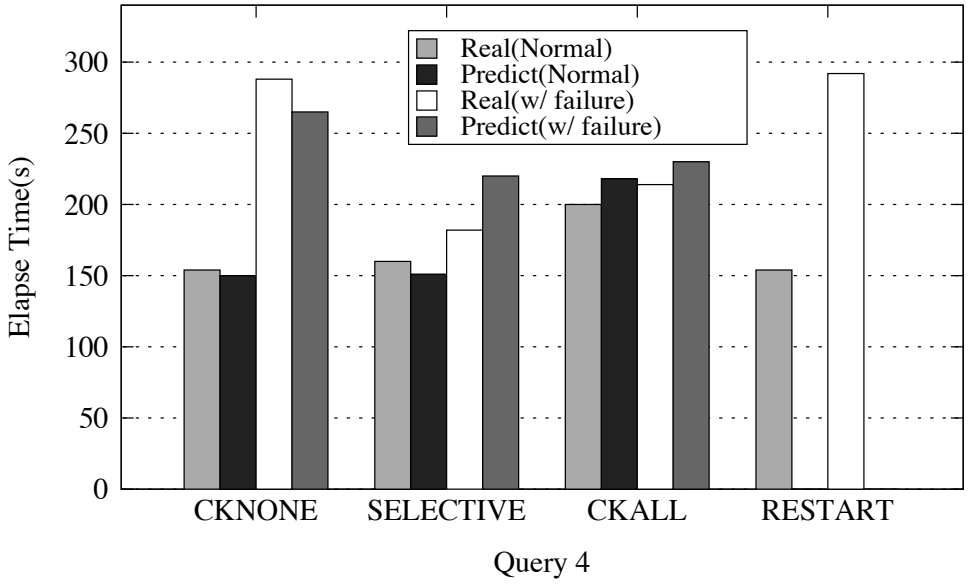

Figure 7.23: Query 4 (SJJJ)

Firstly, SELECTIVE outperforms other strategies for both queries. It produces the smallest slowdown upon a failure with the minimal increase on the execution time with NONE strategy in the normal situation. For Query 4, SELECTIVE increases 3% of the execution time compared to NONE (the best strategy) when no failure occurs while it is 58% faster than NONE when a failure is injected. For Query 5, SELECTIVE is 6% worse than NONE for the execution without a failure but %17 better than NONE in the case of a failure. While SELECTIVE and CKALL both produce several times smaller recovery time than NONE and RESTART, SELECTIVE outperforms CKALL in the overall execution time from 6% to 44%. Specially, although RESTART is at most 5% better than SELECTIVE when no failure occurs, it is 25% and 60% slower than SELECTIVE upon a failure for both queries respectively.

Secondly, SELECTIVE chooses reasonable operators to be checkpointed. For

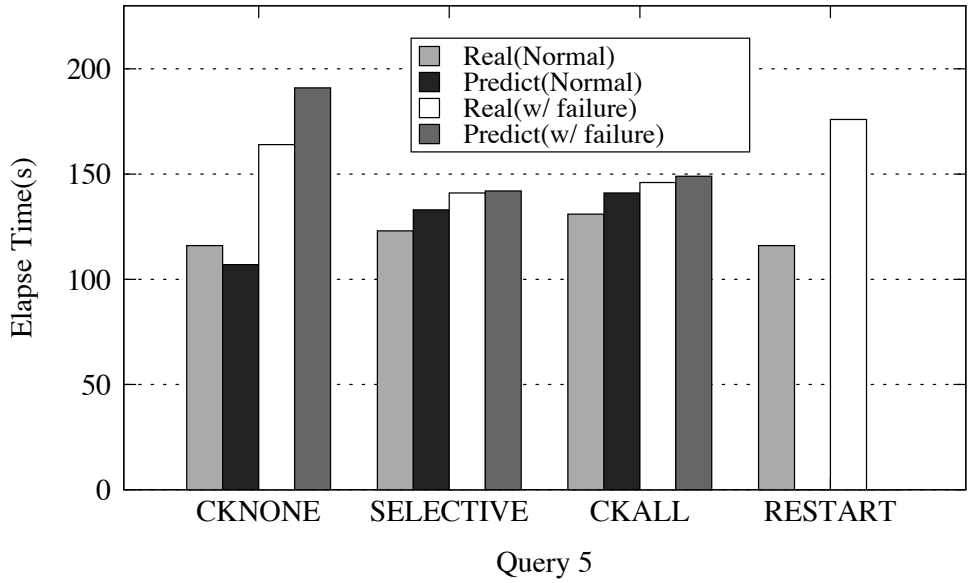

Figure 7.24: Query 5 (SJJG)

Query 4, as the previous operator *J4* produces a large number of output tuples, it is better to checkpoint its predecessor *J2* if it takes a small time and produces small number of output tuples. As we expected, SELECTIVE checkpoints the operator *J2*. Meanwhile, based on the *Sibling Checkpointing Algorithm* described in Section 6.4.2, operator *S1* should be checkpointed too. For Query 5, all operators are checkpointed except *J1*. This is also reasonable because re-producing the output of *J1* is not time-consuming since its two input sources are checkpointed.

7.6.3 Comparison of Slowdown

We conducted a set of experiments to compare the slowdown of Query 5 with ParaLite and Hive[106]. As we know, Hive is a data warehouse system built on top of Hadoop[116]. It translates the HiveQL (SQL-like language) into MapReduce jobs which are executed by Hadoop. Hadoop provides fine-grained fault tolerance because it stores all intermediate data into a durable storage (HDFS). Once a node fails, all failed tasks on that node rather than the whole job are re-scheduled to another node to be executed. ParaLite uses our selective checkpointing mechanism to materialize the output of a set operators and restart all related tasks once a failure occurs.

The results are shown in Figure Fig. 7.25. The X-axis is the percentage of the normal completion time of Query 5 when a failure is injected. Firstly, it is not surprised that ParaLite is about twice faster than Hive. One reason for the

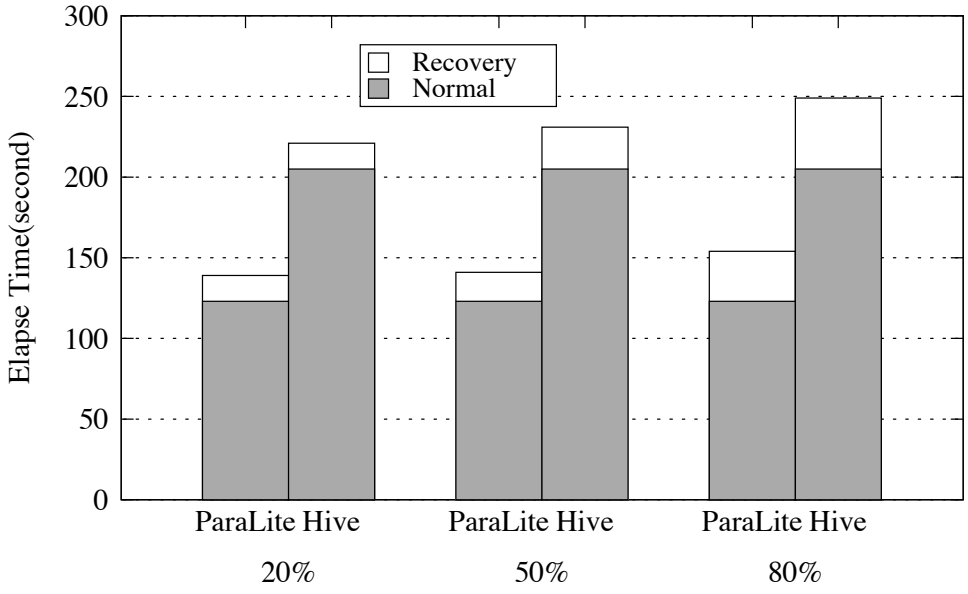

Figure 7.25: The slowdown of Query 5 with ParaLite and Hive

superiority is that not all the operators are checkpointed in ParaLite. Moreover, Hive uses sort-join while ParaLite uses hash-join. In addition, the start-up overhead for Hadoop cannot be ignored. Hive uses 3 MapReduce jobs to express the query and each takes 10 $\sim$ 15 seconds before all map tasks are started. Secondly, the slowdowns of both systems are similar. However, the slowdown of Hadoop is limited to the interval of heartbeat message sent to the JobTracker (master) from each TaskTracker (worker). After a fixed time (we set it 30 seconds) from receiving the last heartbeat, if the master does not receive a new heartbeat message, it sets the status of the worker as failed and re-schedules the failed tasks. Therefore, for a light-weight job whose execution time is not much larger than the interval, the slowdown becomes larger because it needs to wait for the task re-execution. On the other hand, ParaLite detects the failure immediately after a process fails and then starts the recovery.

Chapter 8

Real-world Text Processing Workflows

Contents

In this section, we introduce three real-world text-processing workflows with different structures in natural language processing:

- Japanese Word Count
- Event Recognition
- Sentence Chunking Problem

We compare and discuss the strength/weaknesses both in terms of programmability and performance for each workflow built on top of ParaLite, Hive and Hadoop and Files. Since all these three systems do not provide any language to describe the dependencies of components/jobs, we generally perform each single job using them and leave the creation of the whole workflows to a known workflow engine called GXP Make [103]. GXP Make uses `make` to describe the workflow and provides the parallelization of tasks across clusters. So in the following sections, we ignore the

descriptions of dependencies among jobs and only focus on the expressiveness of each job based on different systems.

8.1 Review of Compared Systems

8.1.1 Hadoop

Hadoop [116] is an open-source incarnation of MapReduce model to process large-scale data across large clusters of compute nodes. It provides users easy programing model by which only two user-customized `Map` and `Reduce` functions are required to be written. Hadoop consists two layers:

(1) Hadoop Distributed File System (HDFS) layer for data storage. HDFS is a block-structured file system which splits individual files into blocks with a fixed size and distributes them across multiple DataNodes in the cluster. HDFS is controlled by a central NameNode which keeps the directory structure of all files in the file system, and tracks the location of blocks and their replicas.

(2) MapReduce layer for data processing. The MapReduce Framework follows the master/worker pattern. A single master called JobTracker receives MapReduce jobs from user applications and schedules tasks to some specific nodes in the cluster determined by the information on NameNode. The policy for job scheduling takes both data locality and load balancing into consideration. Each worker node called TaskTracker accepts tasks including Map, Reduce and Shuffle operations, and spawns separate processes to do the actual work.

Hadoop Streaming (HS) is a utility that comes with the Hadoop distribution. The utility allows you to create and run map/reduce jobs with any executable or script as the mapper and/or the reducer. For instance, to perform the word count task which is to calculate the occurrences of words from a big text, hadoop streaming uses the following statements:

```
Hadoop jar hadoop-streaming.jar
           -input myInputDir
           -output myOutputDir
           -mapper wc_mapper.py
           -reducer wc_reducer.py
```

In the above example, both the mapper and the reducer are python executables. The mapper reads the input from stdin (line by line), splits the input into words and emits the output of ⟨`word, 1`⟩ to stdout. The reducer reads the output of the mapper from stdin and calculates the total number of occurrences for each word. HS

creates a map/reduce job, submits the job to a cluster, and monitors the progress of the job until it completes.

8.1.2 Hive

Hive is a data warehouse system built on top of Hadoop. It is considered as a hybrid of MapReduce model and database system since it projects structured data files to relational database tables and supports queries on the data. These queries are expressed in a SQL-like declarative language called HiveQL and compiled into MapReduce jobs executed on Hadoop. Meanwhile, Hive also allows users' own mappers and reducers which are executables written in any language to be plugged in the query when it is inconvenient or inefficient to express the logic in HiveQL.

With Hive, we can express the word count task by the following query:

```
select mapout.word, count(*) from (
map text using 'wc_mapper.py' as word
from} data) mapout group by mapout.word
```

As Hadoop does, it firstly splits text into words using a nested query in which the executable *wc_ mapper.py* is specified as a mapper and outputs an intermediate table *mapout* of words. Then the outer query aggregates the occurrence of each word using *group by* operation. However, although HiveQL is similar with general SQL and targets to achieve SQL compatibility, it still introduces significant new syntax to normal SQL to integrate MapReduce scripts; for instance, in addition to the usual SELECT, it adds MAP, REDUCE and TRANSFORM.

8.2 Japanese Word Count

`Japanese Word Count` calculates the occurrence of Japanese words from crawled Japanese web pages. Word count task is widely used to extract key words or phrases from web data which is very useful in the web analysis of various fields, such as, revealing hot topics in Twitter, popular products in on-line stores and attracting customs in different countries.

As Fig 8.1 shows, this workflow is a simple pipeline with four jobs: (1) *html2sf*: convert the source data crawled from Japanese web pages to an XML-based canonical format developed by Kurohashi group at Kyoto University. The format defines XML tags to identify plain text and accommodates annotations such as named entities. (2) *sf2rs*: extract plain text, identified with tag ⟨raw_string⟩ from the canonical

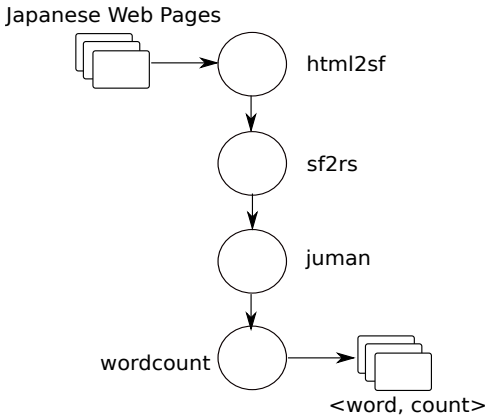

Figure 8.1: Workflow of Japanese Word Count

format data, (3) *juman*: tokenize text into Japanese words, (4) *wordcount*: calculate the frequencies of Japanese words.

Hadoop: The first two jobs are expressed by two Hadoop Streaming[54] commands each of which only contains a mapper *html2sf* or *sf2rs*. To reduce redundant IO operations, we perform the last two jobs together using one command with a mapper *juman* and a reducer *word_ count_ reducer*. The mapper parses Japanese sentences into words and outputs each word with an associated count of occurrences 1 and then the reducer aggregates all counts emitted for a particular word.

The first executable *html2sf* is a file-based program which reads input data from file, so a wrapper which receives data from standard input and stores them into a file is necessary. Moreover, since the input of each executable has multiple lines per record, we have to either customize our own *InputFormat* and *InputReader* classes and pack them along with the streaming jar or write a small wrapper to convert the complex record into a single-line one. In this paper, we take the latter method. Therefore, 3 wrappers for executables are necessary.

Hive: Hive performs the workflow by only one query

```
insert overwrite table wordcount
select tokens.word, count(*) as count from (
   map rst.rs using 'juman' as word from (
      map sft.sf using 'sf2rs' as rs from (
         map html.con using 'html2sf\_wrap' as sf from html)
            sft) rst) tokens group by tokens.word
```

in which the first three executables are nested and the output Japanese words from *juman* are aggregated. To deal with file-based executable *html2sf*, we still need

a wrapper to produce the input file with data from standard input. Since data are piped between executables and each one can handle the output data from the previous one, we don't need to write any wrapper to deal with the complicated format of data.

ParaLite: The first three jobs are expressed by a single query where the executables are specified as three UDXes and data are piped from one to another. The whole workflow is described in Appendix A.1.

```
create table tokens as
select T(S(H(con))) as word from html
with H="html2sf html_file" input 'html_file' S="sf2rs" T="juman"
```

Then another simple SQL query with *group by* operation aggregates the occurrences of Japanese words. To take advantage of SQLite, the results from the first query are partitioned by words. As a result, ParaLite directly assigns the aggregate query to SQLite engine on each node. ParaLite can support file-based UDX, so no wrapper is required for this workflow and we only need to specify the input option for the executable.

File: In file-based workflows, the first three steps are expressed simply by a command line. However, to parallelize the command line, it is necessary to split the big input file into many small files. As a result, many intermediate files are produced. Specially, since there is no straightforward method to express the last aggregation job *wordcount*, we perform it by Hadoop Streaming.

Discussion: Japanese Word Count is a simple pipeline workflow which is expressed by Hive and ParaLite elegantly where more than one executables can be expressed within a single query. However, Hive needs more efforts to deal with file-based executable. Since Hadoop Streaming cannot support multiple mappers or reducers in a single HS job, the executables have to be expressed by several separate HS jobs, leading to a), more steps in the workflow, b), more efforts to deal with the complicated format of input data, c), longer execution time due to storing the output of each executable in files. General jobs with executables are easily expressed with file system, but the aggregate job cannot be presented straightforwardly. Users have to either write their own processing logic or rely on Hadoop. Besides, a large number of intermediate files are produced for the parallelization. The summary for the efforts made by each system is shown in Table. 8.1.

	Hadoop	Hive	ParaLite	File
# of intermediate files	No	No	No	1000
# of wrapper	3	1	0	0

Table 8.1: Comparsion of Productivity of Systems for JAWC

8.3 Event Recognition Application

The goal of *Event-Recognition* [72][9] workflow is to recognize complex bio-molecular relations (bio-events) among biomedical entities (i.e. proteins and genes) that appear in biomedical literature. Recognition of such events including an expression of a certain gene, a phosphorylation of a protein, and a regulation of certain reactions are important to understand biomedical phenomena. The process to extract the events from an English sentence is show in Fig. 8.2.

We hypothesized that the phosphorylation of TRAF2 inhibits binding to the CD40 cytoplasmic domain.

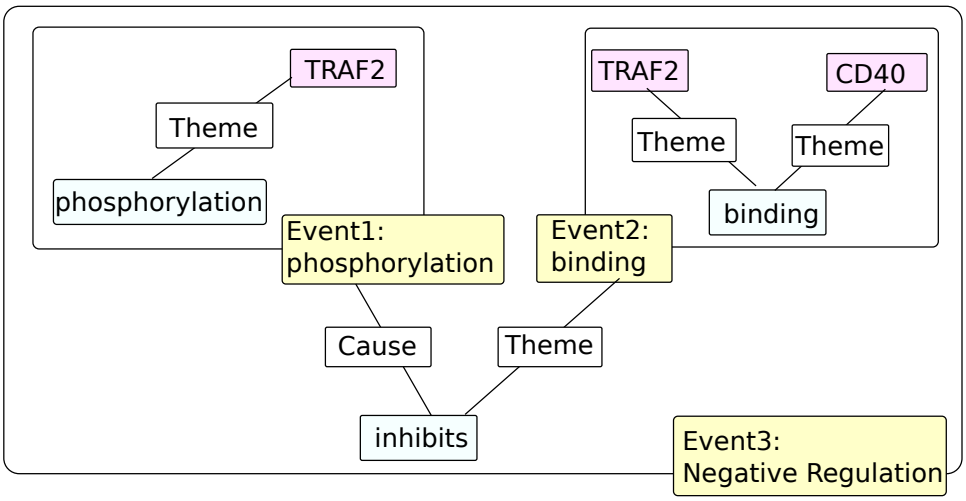

Figure 8.2: Extracted Events from An English Sentence

The workflow of Event Recognition is shown in Fig 8.3. The input of the workflow is the MEDLINE database [41] which contains over 19 million references to journal articles in life sciences with a concentration on biomedicine. The event recognition application consists of 4 steps with 6 jobs: (1) extract abstract of each article from the source xml files; (2) split the abstract into sentences with their unique identification; (3) to each sentence, apply three tools:

- Enju Parser: a HPSG parser which can effectively analyze syntactic/semantic structures of English sentences.

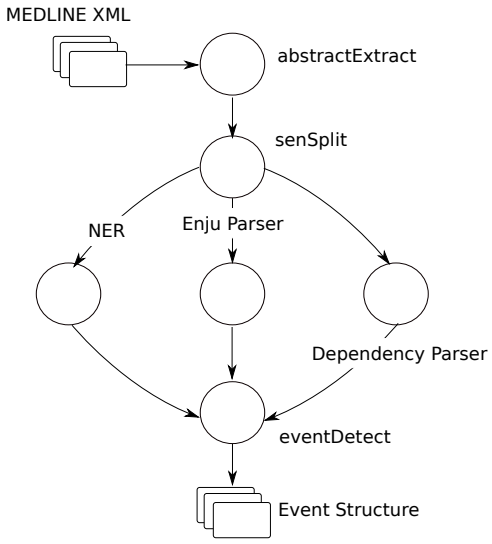

Figure 8.3: Workflow of Event-Recognition Application

- Named Entity Recognizer: recognition for bio-medical entities such as gene and protein.
- Dependency Parser: a dependency parser for biomedical text.

(4) combine the results from the three tools and extract bio-medical events. It is a typical real NLP workflow, which applies several existing tools to each document/sentence and combines results from them to perform a higher-level reasoning. A recurring problem in such workflows is that each tool reads texts as a single stream and does not have a notion of document boundaries. The output from such a tool is similarly a single stream that does not leave anything between document boundaries. Thus, it is the responsibility of workflow developers to track the association between a document and a result from each tool and correctly combines them.

Hadoop: Each job in the workflow could be expressed by one or several Map-Reduce jobs. Generally, tools used in the steps (3) and (4) consist of several executables and some of them work on the joined data from other two previous executables. Hadoop performs join operation separately before the executable is executed and uses several scripts for these executables. For example, the final *eventDetector* job which joins data from the previous three tools on the sentence ID to detect complex relations between entities is expressed by two MR jobs. The first one only performs the join operation using both Map and Reduce functions and outputs records each of which consists of a sentence ID followed by the sentence and the three result of this sentence. Then the next MR job specifies the executable as a mapper which reads output from the previous job and emits the final results.

Hive: Hive expresses each job with one or several equivalent queries. Different from Hadoop, Hive is able to chain several executables or express them together with a join operation within a single query. For example, the `eventDetector` is expressed by the following query:

```
insert overwrite table event_so
select out.SID, out.event
from   (map  abst.SID, abst.sentence, enju_so.enju, ksdep_so.ksdep,
             gene_so.gene using 'event-detector' as (SID, event)
        from abst join enju_so on (abst.SID = enju_so.SID)
                  join ksdep_so on (abst.SID = ksdep_so.SID)
                  join gene_so on (abst.SID = gene_so.SID)) out
```

ParaLite: ParaLite expresses each job by a similar query as Hive. Still taking `eventDetector` as an example, it is expressed by the following query (the whole workflow is shown in Appendix A.2):

```
create table event_so as
select F(abst.SID, abst.sentence, enju_so.enju, ksdep_so.ksdep,
         gene_so.gene) as (SID, event)
from abst, enju_so, ksdep_so, gene_so
where abst.SID = enju_so.SID
      and abst.SID = ksdep_so.SID
      and abst.SID = gene_so.SID
with F="event-detector" output_row_delimiter EMPTY_LINE
```

File: Similar with JAWC application, the input file is firstly split into thousands of small files and several executables are applied to each single file. Specially, for all the merge jobs, we take a in-order processing method, that is, all data are stored in the same order on the sentence ID. To fulfill this requirement, we define the name of each result file before the execution of the workflow.

Discussion: The workflow of Event Recognition generates both data access patterns of pipeline and reduce. Hadoop Streaming and file-based method are not sufficient to present join job. Hive and ParaLite are able to use queries to express the workflow elegantly. However, some extra efforts are necessary when the workflow is performed by Hive and Hadoop because they cannot track the association of the input sentence and the output from the NLP tools as we mentioned in the beginning of this section.

For example, let's say we have an executable X that reads sentences and outputs annotated sentences. In the workflow using such a tool as a component, we like to find (document id, annotated sentence) from (document id, the original sentence). In Hive and Hadoop, it is necessary to write an extra program which extracts sentences fed to the tool, receives the results and maps the annotated sentence to the original one. This is because that MapReduce programming model leaves all the computation inside of the mapper and reducer and it cannot handle complex logical processing outside. Specifically, the model reads data from HDFS, feeds them to a mapper, shuffles and sorts the output of the mapper and finally gives to a reducer. So it doesn't have any mechanism to do some complex processing to the output of mapper of reducer. Hence we need to write ten such wrappers in total. On the other hand, ParaLite, or SQL for that matter, naturally supports such an association through a simple query of the form "select sentence_id,X(sentence) from ...", as long as the output of the last executable in the chain has a fixed string , such as an empty line in most cases, between records boundaries. The summary for the efforts made by each system is shown in Table. 8.2.

	Hadoop	Hive	ParaLite	File
# of intermediate files	No	No	No	50000
# of wrapper	12	10	5	10

Table 8.2: Comparison of Productivity of Systems for ER

8.4 Sentence Chunking Problem

Splitting sentences into meaningful chunks or phrases (N-grams) is very important in natural language processing since it is the first step of extracting concepts and relations within statements across a large text. Significant chunks would typically correspond to semantic units such as named entities (proteins, genes, diseases) or relations [46][89]. A recent paper [4] focused on the improvement of the performance of this application based on MapReduce through a proposed distributed looking-up system.

- phrase-generator: Generate N-grams (phrases) for each sentence.
- phrase-frequency: Calculate the frequencies of phrases.
- db-load: store all phrases whose frequencies are larger than 1 into a SQLite database which is located in a shared file system.

- bf-producer: construct BloomFilter (BF) [6] for phrases whose frequencies are larger than 1. The BF is created to reduce the latency of a look up to the SQLite database. For example, if we want to query the frequency for word `A`, firstly, we check if `A` is in the BF. If not, it means the frequency of this word is 1. Otherwise, another query to the SQLite database is required to get the exactly frequency. Since the latency to look up in a BF is much smaller than SQLite and phrases whose frequencies equal to 1 take a great part in the whole phrases, BF provides a significant improvement of performance.
- frequency-count: Calculate the number of phrases group by their frequencies.
- likelihood-prod: Calculate the likelihood of each sentence.

The problem for Sentence Chunking is to find the best way to chunk a sentence with the most meaningful phrases. We use a statistical model to solve it. Every sentence is generated by randomly sampling and the number of ways to chunk a sentence into phrases is finite. The model calculates the likelihood of each sentence by:

$$L(S) = \sum_{\sigma \in \Phi} \prod_{i \in \sigma} f_i \tag{8.1}$$

Φ is the set of the chunked sentence by all chunking methods; σ represents all phrases in the sentence under a specific chunking method and f_i is the probability of phrase i occurs in a corpus calculated based on its frequency.

The likelihood of whole corpus is simply calculated by the multiplication of the likelihood of each sentence:

$$L(C) = \prod_{S_i \in C} L(S_i) \tag{8.2}$$

We can then maximize the likelihood function of the whole corpus to get the best parameter f.

$$f = \underset{f}{\operatorname{argmax}}\ L(C) \tag{8.3}$$

Then, based on the best value of f, we can calculate the likelihood of each sentence in different chunking methods and the best one leads to the largest likelihood of the sentence.

The workflow of Sentence Chunking is shown in Fig 8.4. The input of the workflow is sentences of articles from the MEDLINE database. The workflow consists of five steps. The first three are required for initialization and only run once per input corpus: (1) `senSplit`: Extract text from input xml files and splitting into sentences, (2) `freqGen`: Generate phrases from sentence and get their frequencies, (3) `filter`: Store phrases with frequencies greater than one into a SQLite database

to reduce the size of the database because most phrases occur only once according to the paper [4]. So receiving NULL for a frequency lookup query means that the frequency for the phrase is one. The following two steps are iterated: (4) `probGen`: newly estimated model parameter f is updated at the end of each iteration if a better distribution is found, (5) `likelihoodCal`: calculate the likelihood of the whole corpus based on the new parameter.

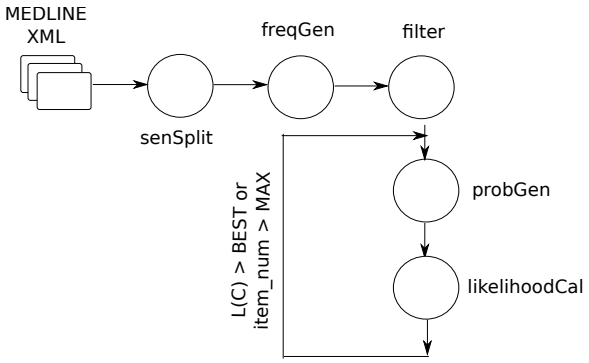

Figure 8.4: Workflow of Sentence Chunking Problem

Hadoop: The `freqGen` job is expressed by a single streaming job in Hadoop in which the mapper executable generates phrases from sentences and emits pairs of ⟨phrase, 1⟩. Then an aggregate reducer reads the output of the mapper and sums the frequency for each phrase. The next `filter` job only requires a mapper which reads phrases from stdin and emits those with frequencies greater than one. Phrases that passed the filter are stored in SQLite database which is queried against those phrase frequencies at each iteration. Next, we calculate the probability of phrase based on their frequencies, so we firstly have another aggregate job to obtain the number of phrases with the same frequency and then a script is invoked to get the related probability. Finally, the last job specifies an executable as the mapper which outputs the likelihood of each sentence to the reducer to get the likelihood of the whole corpus by multiplying likelihood of all sentences.

Hive: Hive uses equivalent streaming operations in a query to express each job, such as `freqGen` and `likelihoodCal` job. Specially, it performs `filter` job by a simple selective query without a customized mapper.

ParaLite: Similar with Hive, ParaLite expresses each job by an equivalent query in which UDXes are used instead of mappers. Specially, for the last job, ParaLite supports a user-defined aggregation `mul` to get the product of all records. The whole workflow is shown in Appendix A.3.

File: In this workflow, aggregate jobs such as `freqGen`, `probGen` and `likelihoodCal` appear alternately. These jobs cannot be elegantly expressed only

based on file systems and as general we use Hadoop instead. Since most of jobs are performed in Hadoop style, we finally did not develop the whole workflow based on files.

Discussion: Each iteration of this workflow is a simple pipeline which is easily expressed by Hadoop, Hive and ParaLite. Although Hadoop provides an overall elegant expression, it still requires more efforts (an extra mapper or reducer) to perform data selection and aggregation. In addition, file-based method is not appropriate for such workflow in which most jobs perform aggregations to all data. The total number of programs that are developed for this workflow are shown in Table. 8.3.

	Hadoop	Hive	ParaLite
# of wrapper	10	5	4

Table 8.3: Comparison of Productivity of Systems for Sentence Chunking

8.5 Evaluation

We conducted several experiments to compare the performance of the three workflows built on top of Hadoop Streaming, Hive, ParaLite and a shared file system in a cluster of 32 nodes. Each node uses 2.40 GHz Intel Xeon processor with 8 cores running 64-bit Debian 6.0 with 24GB RAM.

8.5.1 System Configuration

Hadoop: In our experiments, we use Hadoop version 1.0.3 running on Java 1.6.0. We deploy the system on the cluster with the default configuration settings except for (1) we configure the system to run six Map instances and six Reduce instance concurrently on each node. The reason we set them to be six is that Hive often uses a single query with 2∼3 executables to perform a job, that is, 2∼3 processes are running for each Map task and we observed that Hive cat get the best performance with this configuration. (2) we allow JVM to be reused by all tasks instead of starting a new process for each Map/Reduce task. The number of mappers is decided by the system for most jobs while set manually for some time-consuming jobs to make sure that the execution time of each job is no more than 10 or 30 minutes. Since some jobs have large start-up cost, we do not limit the execution time within 1 minute as Hadoop suggested. To make the comparison fair, we store all input and output data in HDFS with the settings of one replica per block and without compression.

Hive: We use Hive version 0.8.1 with default configuration based on the Hadoop system configured as mentioned above. We set the same number of mappers and

reducers for each Hive job and Hadoop job.

ParaLite: ParaLite is a serverlessness and zero-configuration system, so we do not need to configure anything before it is executed. Each table is stored in the same 32-node cluster and we start at most 6 clients on each node for each job. We set the size of blocks which is sent from data node to the computing clients within each collective query to make sure the total number of blocks equals to the number of mappers for the equivalent Hadoop job.

File: In file-based workflows, we use a shared file system NFS3 to store and transfer data. The parallelism of each job depends on the number of input files N which is determined by the parallel granularity of the most time-consuming job in other systems. For example, N should be equal to the number of mapper tasks for the job in Hadoop.

8.5.2 Japanese Word Count

We perform the experiments for Japanese Word Count workflow with a collection of Japanese web pages of size 104 GB. These web pages produce 62 GB useful text which are then loaded into different systems. The data loading time for each system is shown in Table 8.4.

Hadoop	Hadoop(parallel)	Hive	Hive(parallel)	ParaLite	File
1280	126	1310	131	432	980

Table 8.4: Data Preparation Time for JAWC(sec)

Hadoop: Since we do not need to alter the input data, we load the input file into HDFS as plain text using the Hadoop command-line utility. The input data is a single 62 GB file. If we directly invoke the command line to store it into HDFS from a node, a copy of the file is loaded to one HDFS data node. Another choice is that we split the big file into 32 small ones and store each one in the local disk of each data node respectively, then we load all local files in parallel into HDFS by issuing the command on each node. We measure these two methods and the results are presented in Table 8.4 where label `Hadoop` indicates the first method while `Hadoop (parallel)` means the latter. Obviously, loading data in parallel reduce a lot time but it brings file split overhead.

Hive: Hive can load data to table from both local disk and HDFS by Hive Data Definition Language (DDL). Hive firstly copies it into HDFS and then creates metadata for the table. Since the metadata creation cost is negligible, so it takes almost the same time with Hadoop for the two cases.

ParaLite: ParaLite provides the same API with SQLite and loads data to the

database by the `.import` command line. Unlike Hadoop, ParaLite distributes the big file automatically across all data nodes and really loads data to database on each data node in parallel. The process takes about 7 minutes.

File: The input file is stored in NFS, and we do not need to do anything but split it into 1000 sub-files which takes about 16 minutes. 1000 is chosen because Hadoop automatically splits each job into about 1000 tasks to be executed in parallel.

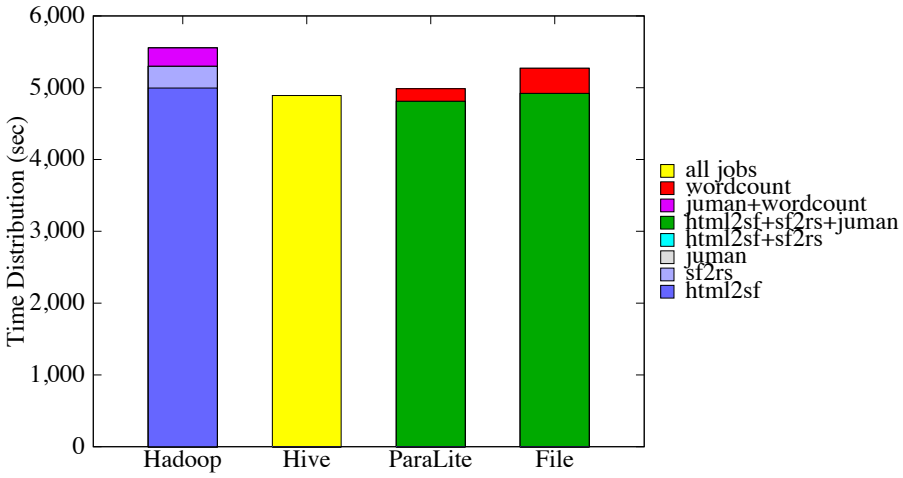

Figure 8.5: The Execution Time of JAWC Workflow

As Fig 8.5 shows, ParaLite and Hive outperforms other systems about 15%. Since Hadoop Streaming cannot pipe multiple executables, three separate Hadoop jobs are launched which brings some extra overheads. For example, the `sf2rs` job is very lightweight and takes less than 2 minutes, but Hadoop Streaming spends 281 seconds on it. One extra overhead comes from the start-up cost of Hadoop. From our observations, it takes 15-25 seconds before all allowed Map tasks have been started. Besides, storing intermediate data (such as 25GB result of `html2sf`) into HDFS which is then read by next process also takes much more time than directly piping data between processes. For the last aggregate job, since the output results from `juman` is partitioned by words, ParaLite executes the SQL query by sending it to the SQLite database on each node and performs local aggregation. Therefore, it outperforms Hadoop since it needs to reduce data.

8.5.3 Event Recognition

Event Recognition workflow reads 30 GB data from MEDLINE database and extracts 1 GB abstracts of articles from the source xml files. We load the data into all systems using the methods mentioned in Section 7.2. For Hadoop and Hive, we directly use the single input file without splitting it and loading from all data nodes

in parallel because the data are small. As a result, Hadoop and Hive take 8 seconds while ParaLite takes 11 seconds. For file-based workflow, we split the input into 10000 small ones according to the most time-consuming job `Enju Parser` and the split takes about 8 seconds.

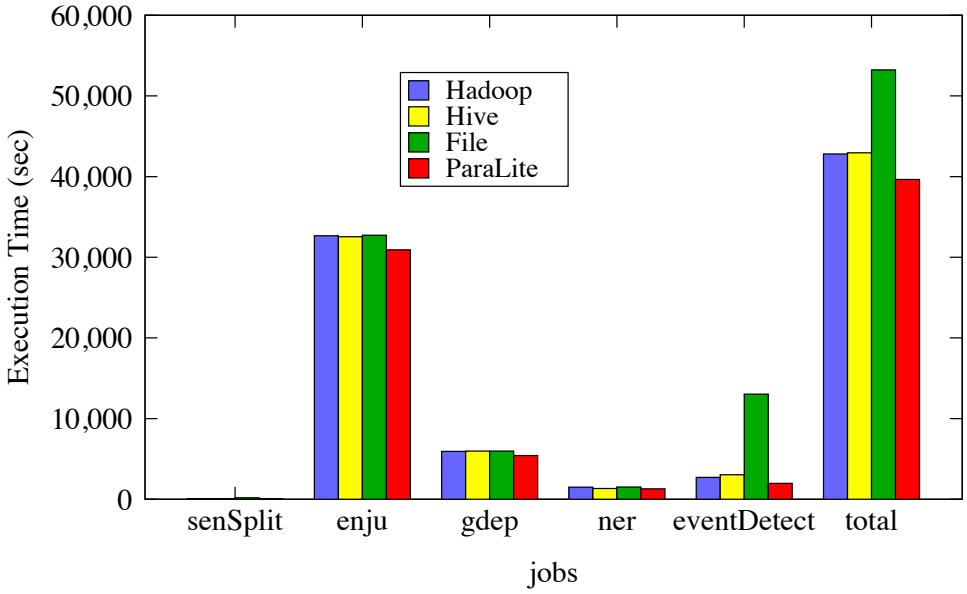

Figure 8.6: The Execution Time of Event-Recog Workflow

Fig 8.6 shows that ParaLite outperforms other systems from 8% to 30%. Since ParaLite is able to track the association of the input and output records, most executables work on the input data directly without parsing while each executable in Hadoop- and Hive-based workflows requires to parse the input data to map the input to the according output. Another reason is that ParaLite has better performance in join operation, especially for the `eventDetector` job. The input four tables of `eventDetector` are 1 GB sentences, 55 GB `enju` results, 11 GB `gdep` results and 150 MB `ner` results. ParaLite partitions all these data on the key `SID`, so when the join operation is performed, it pushes the original join SQL query directly to the SQLite database on each data node and it only takes about 25 seconds. Hadoop performs this join operation using about 8 minutes and Hive takes about 4 minutes.

To get the best performance, we tune some parameters for each job to adjust the degree of jobs parallelization according to their compute density. Job `enju` is very computationally intensive and `eventDetector` is not as heavy as `enju` and it has high start-up overhead, so we set more parallel tasks for `enju` and less for `eventDetector`. It is easy to do the parameter tuning in ParaLite which allows you to specify the size of block for each query and Hadoop which allows you to set the number of mappers and reducers in the script for each job. However, it is

not easy with Hive to tune this parameter. We have to modify the parameter of number of mappers in the configuration file and restart the Hadoop cluster every time when we want to change it. What is the worse is that this kind of parameter tuning is impossible in file-based workflow. This is the reason that the execution time of `event-detector` job in the workflow with files is much larger than that in the workflow with other systems. As mentioned in the beginning of this section, we split the input file into 10000 small ones based on the execution of `enju` job. Hence we have 10000 small sub-jobs to be processed in parallel for each step. The number of sub-jobs is much larger than that in other systems and each has high start-up overhead (about 20 seconds) , as a result, the total execution time is increased. Once the input files is split, users have to parallelize each job according to the number of sub-files unless internal parallelization and merge is performed independently.

8.5.4 Sentence Chunking

Sentence Chunking workflow reads 60 GB data from MEDLINE database and gets 2 GB abstract. We load the data into HDFS using the Hadoop command-line utility and it takes 14 seconds. Hive takes several seconds more to create the metadata and ParaLite takes 21 seconds. From Fig 8.7 we can see that the execution time

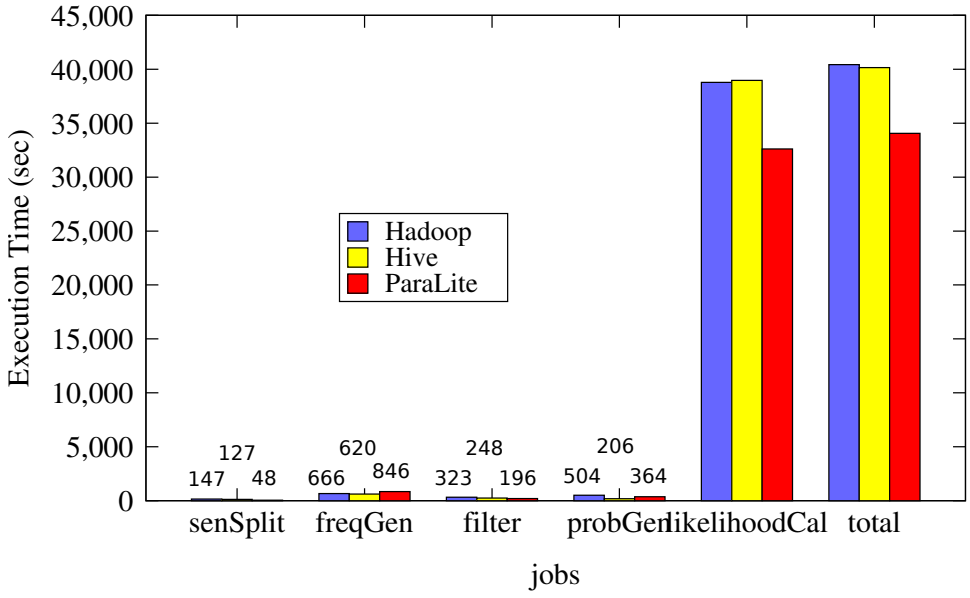

Figure 8.7: The Execution Time of Sentence Chunking Workflow

of the whole workflow by Hive and Hadoop are almost the same and ParaLite is about 18% faster than them mainly because of ParaLite has better performance on the most time-consuming job `likelihoodCal`. For the `freqGen` job, Hadoop and

Hive are about 200 seconds faster than ParaLite. That is because the output of `freqGen` is about 145 GB and storing them into database takes ParaLite another 420 seconds. For the `probGen` job, Hadoop is much more slower than others. The data to be aggregated in this job is very unbalanced and more than 90% phrases (about 137 GB) occur only once. So all these data are transferred to a single reduce task to be aggregated in Hadoop leading to longer execution time than Hive and ParaLite that firstly aggregate data locally.

Chapter 9

Conclusions

Contents

9.1 Conclusions

This thesis proposes ParaLite – a shared-nothing parallel database system for data-intensive workflows. For general SQL queries, ParaLite has similar performance with a commercial database system and several times outperforms Hive. With ParaLite, jobs in a workflow are expressed with SQL queries and all intermediate data are stored as relational tables. To facilitate the workflow's description and increase the performance of its execution, ParaLite provides the following specific features for workflows.

As workflows are typically built out of various executables, ParaLite provides seamless integrations of external executables (UDX, short for User-Defined Executable) into SQL statements and proposes a concept of *collective query* for the efficient parallel execution of UDX through co-allocation of parallel compute clients and data sources the two driving design principles. With the support of UDX, users do not need to write any programs that are conformed to strict specifications of databases and are not constrained in the languages they can use. The UDX implementation does not invoke the executable on every single tuple, making ParaLite 10x speed up comparing to UDF implementation in the conventional database. The experimental results also show that with collective queries the performance for the UDX's execution could achieve close-to-ideal speedup with the increase of computing clients when data are either balanced or not balanced distributed across a cluster. In addition, the mechanism of collective query adapts the load across computing clients even when some clients are manually loaded.

For long-running jobs in a workflow, ParaLite supports intra-query fault tolerance with a selective checkpointing mechanism, enabling to resume queries from

the middle of the execution upon a failure. The mechanism models the cost of a query execution with the consideration of fine-grained parallelism and takes a divide-and-conquer algorithm with the complexity of O(n) to select a set of operators whose outputs are worth being checkpointed. The experimental results firstly show that different fault-tolerant strategies affect the overall runtimes of queries. Then our selective checkpointing can choose reasonable operators to be checkpointed and outperforms other fault-tolerant strategies, such as simple pipelining data and checkpointing all intermediate data. In addition, the divide-and-conquer algorithm taken by our mechanism has a smaller overhead than brute-force approach while keeping a similar effectiveness.

Finally, we studied three real-world text-processing workflows, specifically in the discipline of Natural Language Processing (NLP), and built them on top of ParaLite, Hadoop, Hive and general files. We discuss their strengths/weaknesses both in terms of programmability and performance for each workflow. Our development experience revealed that high-level query languages such as SQL of ParaLite and HiveQL of Hive are helpful for expressing data selection, aggregation and calculation by typical executables. In NLP workflows, the expressiveness of SQL in ParaLite is particularly useful since it provides natural supports of file-based NLP executables and reusing existing NLP tools by tracking the association between a document and its annotation attached by the tool. On the other hand, workflows expressed in low-level language lack good support of all features mentioned above, requiring some extra efforts. The experimental results show that essentially each system has similar performance in the execution of the whole workflows because performing executables takes most time. However, the small differences still revealed some potential superiority of ParaLite due to data partitioning and query optimization.

9.2 Future Work

The major future work of ParaLite includes performance improvements and functional enhancement to make ParaLite more productive for workflows. As the performance analysis for TPC-H queries shows, one potential bottleneck for ParaLite is that a complex query is re-written to multiple simpler queries and it suffers from bad performance for storing intermediate data of each query into databases. Thus, more sophisticated query planner should be implemented to release ParaLite from this limitation. Another work exists in eliminating the limitation that ParaLite cannot organize data on a particular columns to provide the partial order when the data are loaded into the database. In addition, we would like to reduce the latency to query ParaLite database. As most NLP executables have high start-up overhead,

it is difficult to return the results within a short latency if such executables are spawned after the system receives the query. A possible solution for this problem is to make the executables stay running on some resources and data are transferred to the nodes which are running the processes.

Functional enhancement to make ParaLite more productive for workflows is another important future work. Firstly, a more general and user-friendly method of describing various formats from external executables is necessary. Current UDXes support the executables with simple formats of input and output where records/-columns are distinguished by a specific string. To support executables with more complex formats, one solution is to define the specification with regex, BNF or Xpath rather than the simple string. Secondly, detection and notification to users of error data for an executable are general problems in workflows. Sometimes, a failure of an UDX is caused by receiving error data and cannot be recovered by re-execution. The system should detect this kind of failure and notify the user the probable problem data.

Appendix A

Workflows Description

A.1 Japanese Word Count Workflow with ParaLite

```
all: $(WORD_COUNT)

$(HTML_FILE) : $(INPUT_FILES)
	./readcrawl.py $< > $@
$(HTML_TABLE) : $(HTML_FILE)
	paralite DB "create table html(rowid, rar,url,rbt,res,tim,req,
	    sta,hdr,con) on DATA_NODE"
	paralite DB ".import $< html -column_separator ::::: -
	    row_separator ====="
$(TOKENS) : $(HTML_TABLE)
	paralite DB "create table tokens as select T(S(H(con))) as
	    word from html with T="juman" S="sf2rs" H="html2sf
	    html_file" input 'html_file' on $(DATA_NODE) partition by
	    word collective by 1"
$(WORD_COUNT) : $(TOKENS)
	paralite DB "create table word_count as select word, count(*)
	    as frequency from tokens group by word on DATA_NODE"
```

A.2 Event Recognition Workflow with ParaLite

```
ALL: $(EVENT_SO)

$(ABST_TXT) : $(PUBMED_XML_GZ)
	zcat $^ | xml2text '###' > $@
	paralite DB "create table abst_txt(PMID, abstract) on
	    DATA_NODE"
	paralite DB ".import $@ abst_txt"
$(ABST_SS) : $(ABST_TXT)
	paralite DB "create table abst_ss as select F(PMID, abstract)
	    as (SID, sentence) from abst_txt with F="geniass_wrap"
	    input_col_delimiter '###' output_col_delimiter '###' on
	    DATA_NODE partition by SID collective by 1"
$(ENJU_SO) : $(ABST_SS)
```

```
	paralite DB "create table enju_so as select SID, A(E(sentence)
	    ) as enju from abst_ss with E="enju" A="add_lex_head_wrap"
	    output_row_delimiter '====####====' on $(DATA_NODE)
	    partition by SID collective by 2"
$(GDEP_OUT) : $(ABST_SS)
	paralite DB "create table ksdep_out as select SID, F(sentence)
	     as pre_ksdep from abst_ss with F="gdep"
	    output_row_delimiter EMPTY_LINE on $(DATA_NODE) partition
	    by SID collective by 3"
$(GDEP_SO) : $(GDEP_OUT)
	paralite DB "create table ksdep_so as select abst_ss.SID as
	    SID, F(abst_ss.sentence, ksdep_out.pre_ksdep) as ksdep from
	     abst_ss, ksdep_out where abst_ss.SID = ksdep_out.SID with
	    F="DEP2SO_WRAP -g" input_row_delimiter EMPTY_LINE
	    input_col_delimiter '###' output_row_delimiter EMPTY_LINE
	    on $(DATA_NODE) partition by SID collective by 4"
$(GENE_NE_TEMP) : $(ABST_SS)
	paralite DB "create table gene_ne_temp as select SID, G(T(
	    sentence)) as pre_gene from abst_ss with T="
	    gene_ner_tokenizer /dev/stdin" G="gene_ner_gtag off /dev/
	    stdin /dev/stdout" output_row_delimiter EMPTY_LINE on $(
	    DATA_NODE) partition by SID collective by 5"
$(GENE_NE_SO) : $(GENE_NE_TEMP)
	paralite DB "create table gene_ne_so as select abst_ss.SID as
	    SID, F(N(M(abst_ss.sentence, gene_ne_temp.pre_gene))) as
	    gene from abst_ss, gene_ne_temp where abst_ss.SID =
	    gene_ne_temp.SID with M="pos_maker_wrap /dev/stdin 0"
	    input_row_delimiter EMPTY_LINE input_col_delimiter '###' N=
	    "gene_ner_dict_matcher /dev/stdin" F="ner_wrap /dev/stdin"
	    output_row_delimiter EMPTY_LINE on $(DATA_NODE) partition
	    by SID collective by 6"
$(EVENT_SO) : $(GENE_NE_SO) $(GDEP_SO) $(ENJU_SO) $(ABST_SS)
	paralite DB "create table event_so as select F(abst_ss.SID,
	    abst_ss.sentence, enju_so.enju, ksdep_so.ksdep, gene_ne_so.
	    gene) as (SID, event) from abst_ss, enju_so, ksdep_so,
	    gene_ne_so where abst_ss.SID = enju_so.SID and abst_ss.SID
	    = ksdep_so.SID and abst_ss.SID = gene_ne_so.SID with F="
	    event_detector $(EVR_JAVA) medie $(EVR_MODEL)/ $(FILE_ID)"
	    input_row_delimiter EMPTY_LINE input_col_delimiter '###'
	    output_row_delimiter '====' on $(DATA_NODE) collective by 6
	    "
```

A.3 Sentence Chunking Workflow with ParaLite

Makefile.pre:

```
ALL: $(FREQUENCY_COUNT) $(BLOOMFILTER) $(SQDB)

$(ABST_TXT) : $(PUBMED_XML_GZ)
	zcat $^ | xml2text '###' > $@
	paralite DB "create table abst_txt(abstract) on DATA_NODE"
	paralite DB ".import $@ abst_txt"
$(ABST_SS) : $(ABST_TXT)
	paralite DB "create table abst_ss as select G(F(abstract)) as
	    sentence from abst_txt with G="tokenize /dev/stdin" F="
	    geniass INPUT OUTPUT" input 'INPUT' output 'OUTPUT)' on
	    DATA_NODE collective by 1"
$(N_GRAM) : $(ABST_SS)
	paralite DB "create table n_gram as select G(sentence) as
	    phrase from abst_ss with G="n_gram_splitter MAX_LENGTH" on
	    DATA_NODE partition by phrase collective by 1"
$(PHRASE_FREQUENCY) : $(N_GRAM)
	paralite DB "create table phrase_frequency as select phrase,
	    count(*) as frequency from n_gram group by phrase on
	    DATA_NODE"
$(SMALL_PHRASE_FREQUENCY) : $(PHRASE_FREQUENCY)
	paralite DB "select phrase, frequency from phrase_frequency
	    where frequency > 1" > $@
$(SQDB) : $(SMALL_PHRASE_FREQUENCY)
	sqlite3 SQDB "create table phrase_frequency (phrase varchar
	    (100), frequency int)"
	sqlite3 SQDB ".import $^ phrase_frequency"
$(BLOOMFILTER) : $(SMALL_PHRASE_FREQUENCY)
	./bf_producer SIZE NUN_HASH $^ | $@
$(FREQUENCY_COUNT) : $(PHRASE_FREQUENCY)
	paralite DB "select frequency, count(*) from phrase_frequency
	    group by frequency"
```

Makefile.loop:

```
ALL: $(CORPUS_LIKELIHOOD)

$(FREQUENCY_PROBABILITY) : $(FREQUENCY_COUNT)
	$(PROBABILITY_PRODUCER) $^ $@

$(SENTENCE_LIKELIHOOD) :
	time paralite $(DB) "create table sentence_likelihood_1 as
	    select F(sentence) as hood from abst_ss_1 with F="
	    likelihood_producer.py $(BLOOMFILTER) $(SIZE) $(NHASH) $(
	    CACHE_SIZE) $(SQDB) $(FREQUENCY_PROBABILITY) ### " on $(
	    DATA_NODE) collective by 1"

$(CORPUS_LIKELIHOOD) : $(SENTENCE_LIKELIHOOD)
```

```
        paralite $(DB) "select sum(likelihood) from
            sentence_likelihood" > $@

best:
        cp -f $(FREQUENCY_PROBABILITY) $(BEST_PROBABILITY)
```

run.sh:

```
iteration_num=10
max_value=0
i=0

make -f Makefile.pre

while [ $i -lt $iteration_num ]
do
    make -f Makefile.loop
    value = `cat corpus-likelihood.dat`
    if [ $value > $max_value ]; then
        max_value=$value
        make -f Makefile.loop best
    fi
    make -f Makefile.loop clean
    i=$(($i+1))
done
```

Bibliography

[1] Serge Abiteboul, Victor Vianu, Bradley S. Fordham and Yelena Yesha. *Relational Transducers for Electronic Commerce*. J. Comput. Syst. Sci., vol. 61, no. 2, pages 236–269, 2000. 27

[2] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz and Alexander Rasin. *HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads*. Proceedings of VLDB, vol. 2, no. 1, pages 922–933, 2009. 24, 25, 30, 31

[3] *Aster*. http://www.asterdata.com/. 25

[4] Atilla Soner Balkir, Ian Foster and Andrey Rzhetsky. *A Distributed Look-up Architecture for Text Mining Applications using mapreduce*. In Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC11), 2011. 101, 103

[5] Derik Barseghian, Ilkay Altintas, Matthew B. Jones, Daniel Crawl, Nathan Potter, James Gallagher, Peter Cornillon, Mark Schildhauer, Elizabeth T. Borer, Eric W. Seabloom and Parviez R. Hosseini. *Workflows and Extensions to the Kepler Scientific Workflow System to Support Environmental Sensor Data Access and Analysis*. Ecological Informatics, pages 42–50, 2010. 1, 17, 26, 27

[6] B.Bloom. *Space/time Tradeoffs in the Hash Coding with Allowable Errors*. CACM, vol. 13, no. 7, pages 422–426, 1970. 102

[7] Gordon Bell, Tony Hey and Alex Szalay. *COMPUTER SCIENCE: Beyond the Data Deluge*. Science, vol. 323, pages 1297–1298, 2009. 1

[8] D. Bitton and J. Gray. *Disk Shadowing*. In Proc. of VLDB, pages 331–338, 1988. 30

[9] Bjorne, F. Ginter, S. Pyysalo, J. Tsujii and T. Salakoski. *Complex Event Extraction at PubMed Scale*. Bioinformatics, vol. 26, no. 12, pages 382–390, 2010. 98

[10] Anthony J. Bonner. *Workflow, Transactions, and Datalog*. In Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pages 294–305, 1999. 27

[11] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose and Rares Vernica. *Hyracks: A Flexible and Extensible Foundation for Data-intensive Computing*. In ICDE, pages 1151–1162, 2011. 31

[12] *CaboCha Yet Another Japanese Dependency Structure.* `http://code.google.com/p/cabocha`. 3

[13] Michael J. Carey and Laura M. Haas. *Extensible Database Management Systems*. SIGMOD Record, no. 4, pages 54–60, 1990. 29

[14] *Cascading.* `http://www.cascading.org/`. 29

[15] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver and Jingren Zhou. *SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets.* PVLDB, vol. 1, no. 2, pages 1265–1276, 2008. 24

[16] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw and Nathan Weizenbaum. *FlumeJava: Easy, Efficient Data-parallel Pipelines*. In PLDI, pages 363–375, 2010. 25

[17] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon and Michael Wong. *Tenzing A SQL Implementation On The MapReduce Framework*. PVLDB, vol. 4, no. 12, pages 1318–1327, 2011. 24

[18] Surajit Chaudhuri. *An Overview of Query Optimization in Relational Systems*. In Alberto O. Mendelzon and Jan Paredaens, editeurs, Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA, pages 34–43. ACM Press, 1998. 13

[19] S. Chaudhuri and K. Shim. *Optimization of Queries with User-defined Predicates*. ACM Trans. Database Syst., vol. 24, pages 177–228, 1999. 29

[20] Qichang Chen, Liqiang Wang and Zongbo Shang. *MRGIS: A MapReduce-Enabled High Performance Workflow System for GIS*. In eScience, pages 646–651, 2008. 29

[21] Songting Chen. *Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce*. PVLDB, vol. 3, no. 2, pages 1459–1468, 2010. 24

[22] Hung chih Yang, Ali Dasdan, Ruey-Lung Hsiao and Douglas Stott Parker Jr. *Map-reduce-merge: Simplified Relational Data Processing on Large Clusters*. In SIGMOD Conference, pages 1029–1040, 2007. 25

[23] Tyson Condie, Neil Conway, Peter Alvaro and Joseph M. Hellerstein. *MapReduce Online*. In Proc. of the 7th NSDI Symp., 2010. 31

[24] George P. Copeland and Tom W. Keller. *A Comparison Of High-Availability Media Recovery Techniques*. In SIGMOD Conference, pages 98–109, 1989. 30

[25] Microsoft Corporation. *Table-valued User-defined Functions*. Rapport technique, Microsoft, 2009. 29

[26] P. Couvares, T. Kosar, A. Roy, J. Weber and K. Wenger. *Workflows for E-Science: Scientific Workflows for Grids*. Springer Press, 2008. 16, 26, 27

[27] *CPU burn-in*. http://users.bigpond.net.au/CPUburn/. 82

[28] Umeshwar Dayal, Eric N. Hanson and Jennifer Widom. *Active Database Systems*. In Modern Database Systems, pages 434–456, 1995. 27

[29] *IBM DB2 Database Software*. http://www-01.ibm.com/software/data/db2/. 2, 29, 67

[30] *DBInputFormat*. http://www.cloudera.com/blog/2009/03/database-access-with-hadoop/. 26

[31] A. L. Dean, editeur. Proceedings of 1972 acm-sigfidet workshop on data description, access and control, denver, colorado, november 29 - december 1, 1972. ACM, 1972. 10

[32] J. Dean and S. Ghemawat. *MapReduce:Simplified Data Processing on Large Clusters*. In OSDI'04, pages 137–150, 2004. xi, 1, 2, 9, 23, 28, 31, 45

[33] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: a Flexible Data Processing Tool*. Commun. ACM, vol. 53, no. 1, pages 72–77, 2010. 24

[34] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob and Daniel S. Katz. *Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems*. Scientific Programming Journal, vol. 13, no. 3, pages 219–237, July 2005. 1, 26, 27

[35] Ewa Deelman, Dennis Gannon, Matthew S. Shields and Ian Taylor. *Workflows and E-science: An Overview of Workflow System Features and Capabilities*. Future Generation Comp. Syst., vol. 25, no. 5, pages 528–540, 2009. 1

[36] D.J. DeWitt and J. Gray. *Parallel Database Systems: the Future of High-performance Database Systems*. Commun, vol. 35, no. 6, pages 85–98, 1992. 2, 3, 7, 8, 23

[37] D. DeWitt and M. Stonebraker. *MapReduce: A Major Step Backwards*. In The Database Column, 1, 2008. 23, 24

[38] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty and Jörg Schad. *Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without it Even Noticing)*. Proc. VLDB Endow., vol. 3, no. 1-2, pages 515–529, September 2010. 25

[39] *Enju*. http://www-tsujii.is.s.u-tokyo.ac.jp/enju, 2011. 3, 47, 75

[40] Chen et. al. *High Availability and Scalability Guide for DB2 on Linux, Unix, and Windows*. Rapport technique, IBM, 2007. 14

[41] D. FA. *Searching Medline via Pubmed*. In Clin Lab Sci, 2008. 67, 98

[42] Xubo Fei, Shiyong Lu and Cui Lin. *A MapReduce-Enabled Scientific Workflow Composition Framework*. In ICWS '09, pages 663–670, 2009. 28

[43] E Friedman, P Pawlowski and J. Cieslewicz. *SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions*. In Proceedings of VLDB Endow., pages 1402–1413, 2009. 25, 30

[44] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan and Utkarsh Srivastava. *Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience*. In IN VLDB '09: Proceedings of the VLDB endowment, pages 1414–1425, 2009. 2, 24, 29, 30, 31

[45] *GENIA Sentence Splitter*. https://github.com/TsujiiLaboratory/geniass. 45

[46] Sharon Goldwater, Thomas L. Griffiths and Mark Johnson. *Contextual Dependencies in Unsupervised Word Segmentation*. In Meeting of the Association for Computational Linguistics - ACL, 2006. 101

[47] Leana Golubchik and Richard R. Muntz. *Fault Tolerance Issues in Data Declustering for Parallel Database Systems.* Bulletin of the Technical Committee on Data Engineering, vol. 14, 1994. 30

[48] Daniel James Goodman. *Introduction and Evaluation of Martlet: a Scientific Workflow Language for Abstracted Parallelisation.* In WWW, pages 983–992, 2007. 28

[49] Narayan Gowraj, Prasanna Venkatesh Ravi, Mouniga V and M. R. Sumalatha. *S2MART: Smart Sql to Map-Reduce Translators.* In APWeb, pages 571–582, 2013. 25

[50] Jim Gray and Andreas Reuter. Transaction processing: Concepts and techniques. Morgan Kaufmann, 1993. 30

[51] Jim Gray. *Jim Gray on eScience: A Transformed Scientific Method.* In Tony Hey, Stewart Tansley and Kristin Tolle, editeurs, The Fourth Paradigm: Data-Intensive Scientific Discovery, pages xix–xxxiii. 2009. 1

[52] *Greenplum Database.* `http://www.greenplum.com/`. 25, 30

[53] *Hadoop.* `http://hadoop.apache.org/`. 2

[54] *Hadoop Streaming.* `http://hadoop.apache.org/common/docs/r0.15.2/streaming.html`. 96

[55] J. M. Hellerstein and M. Stonebraker. *Predicate Migration: Optimizing Queries with Expensive Predicates.* In Proceedings of SIGMOD Conf., pages 267–276, 1993. 29

[56] Joseph M. Hellerstein and Jeffrey F. Naughton. *Query Execution Techniques for Caching Expensive Methods.* In In SIGMOD, pages 423–434, 1996. 29

[57] T. Hiraishi, T. Abe, Y. Miyake, T. Iwashita and H. Nakashima. *Xcrypt: Flexible and Intuitive Job-parallel Script.* In Symposium on Advanced Computing Systems and Infrastructures (SACSIS2010), pages 183–191, 2010. 27

[58] *Running TPC-H queries on Hive. Web page.* `https://issues.apache.org/jira/browse/HIVE-600`, 2009. 66, 74

[59] H. Hsiao and D. J. DeWitt. *Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines.* In Proceedings of Data Engineering, pages 456–465, 1990. 30

[60] Soonwook Hwang and Carl Kesselman. *GridWorkflow: A Flexible Failure Handling Framework for the Grid*. In Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing, HPDC '03, pages 126–, 2003. 19

[61] Yannis E. Ioannidis, Miron Livny, Shivani Gupta and Nagavamsi Ponnekanti. *ZOO : A Desktop Experiment Management Environment*. In VLDB, pages 274–285, 1996. 27

[62] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell and Dennis Fetterly. *Dryad: Distributed Data-parallel Programs from Sequential Building Blocks*. In EuroSys, pages 59–72, 2007. 25, 27, 31

[63] M. Ivanova, R. Goncalves N. Nes and M. Kersten. *MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database*. In Proceedings of the 19th International Conference on Scientific and Statistical Database Management (SSDBM'07), pages 13–13, 2007. 3

[64] M. Jaedicke and B. Mitschang. *On Parallel Processing of Aggregate and Scalar Functions in Object-relational DBMS*. In Proceedings of SIGMOD Conf., pages 379–389, 1998. 29, 30

[65] Michael Jaedicke and Bernhard Mitschang. *User-Defined Table Operators: Enhancing Extensibility for ORDBMS*. In VLDB, pages 494–505, 1999. 30

[66] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, G. Bruce Berriman, Benjamin P. Berman and Philip Maechling. *Data Sharing Options for Scientific Workflows on Amazon EC2*. In SC, pages 1–9, 2010. 18

[67] Rania Khalaf, Alexander Keller and Frank Leymann. *Business Processes for Web Services: Principles and Applications*. IBM Systems Journal, vol. 45, no. 2, pages 425–446, 2006. 16, 26

[68] Max Kremer, Max Kremer, Jarek Gryz and Jarek Gryz. *A Survey of Query Optimization in Parallel Databases*. Rapport technique, 1999. 13

[69] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He and Xiaodong Zhang. *YSmart: Yet Another SQL-to-MapReduce Translator*. In ICDCS, pages 25–36, 2011. 25

[70] David T. Liu and Michael J. Franklin. *The Design of GridDB: A Data-Centric Overlay for the Scientific Grid*. In VLDB, pages 600–611, 2004. 27, 28

[71] *MEDIE - Semantic Retrieval Engine for MEDLINE.* http://www.nactem.ac.uk/tsujii/medie/. 41

[72] M. Miwa, R. Satre, J.-D. Kim and J. Tsujii. *Event Extraction with Complex Event Classification Using Rich Features.* In JBCB, pages 131–146, 2010. 98

[73] *MySQL Cluster.* http://www.mysql.com/products/cluster/. 30

[74] Phuong Nguyen and Milton Halem. *A MapReduce Workflow System for Architecting Scientific Data Intensive Applications.* In SECLOUD '11, pages 57–63, 2011. 29

[75] María A. Nieto-Santisteban, Jim Gray, Alexander S. Szalay, James Annis, Aniruddha R. Thakar and William O'Mullane. *When Database Systems Meet the Grid.* In CIDR, pages 154–161, 2005. 28

[76] L. E. Notter. *Medline-newest Service in the Medical Information Network.* In Nursing Research, 1972. 67

[77] Thomas M. Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat and Peter Li. *Taverna: a Tool for the Composition and Enactment of Bioinformatics Workflows.* Bioinformatics, vol. 20, no. 17, pages 3405–3054, 2004. 1, 17, 26, 27

[78] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar and Andrew Tomkins. *Pig Latin: A Not-So-Foreign Language for Data Processing.* In International Conference on Management of Data - SIGMOD, pages 1099–1110, 2008. 24, 29

[79] Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki B. N. Rao, Vijayanand Sankarasubramanian, Siddharth Seth, Chao Tian, Topher ZiCornell and Xiaodan Wang. *Nova: Continuous Pig/Hadoop Workflows.* In SIGMOD Conference, pages 1081–1090, 2011. 29

[80] *Apache. Oozie: Hadoop workflow system.* http://yahoo.github.com/oozie/. 29

[81] *Oracle Database.* http://www.oracle.com/. 2, 29, 30, 67

[82] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden and Michael Stonebraker. *A Comparison of Approaches to Large-scale Data Analysis.* In SIGMOD '09: Proceedings of the

2009 ACM SIGMOD International Conference, pages 165–178, 2009. 3, 23, 25

[83] Rob Pike, Sean Dorward, Robert Griesemer and Sean Quinlan. *Interpreting the Data: Parallel Analysis with Sawzall*. Scientific Programming, vol. 13, no. 4, pages 277–298, 2005. 24

[84] K. Plankensteiner, R. Prodan and T. Fahringer. *Fault-tolerant Behaviour in State-of-the-art Grid Workflow Management Systems*. In CoreGRID Technical Report, TR-0091, 2007. 19

[85] *PostgreSQL: the World's Most Advanced Open Source Database*. http://www.postgresql.org/. 25, 26, 30, 31

[86] *PubMed: US National Library of Medicine*. http://www.ncbi.nlm.nih.gov/pubmed. 67

[87] *Sun Grid Engine Home*. http://wikis.sun.com/display/GridEngine/Home, 2011. 26, 27

[88] A. Ray. *Oracle Data Guard: Ensuring Disaster Recovery for the Enterprise*. Rapport technique, Oracle, Mar 2002. 14

[89] Chiara Sabatti and Kenneth Lange. *Genomewide Motif Identification Using a Dictionary Model*. Proceedings of The IEEE, vol. 90, no. 11, pages 1803–1810, 2002. 101

[90] Srinath Shankar, Ameet Kini, David J. DeWitt and Jeffrey F. Naughton. *Integrating Databases and Workflow Systems*. SIGMOD Record, vol. 34, no. 3, pages 5–11, 2005. 28

[91] Takeshi Shibata, SungJun Choi and Kenjiro Taura. *File-access Patterns of Data-intensive Workflow Applications and their Implications to Distributed File Systems*. In HPDC, pages 746–755, 2010. xi, 16

[92] *SQL Server*. http://www.microsoft.com/en-us/sqlserver/default.aspx. 2, 28, 67

[93] *SQLite*. http://www.sqlite.org/. 33

[94] Michael Stonebraker, Jeff Anton and Eric N. Hanson. *Extending a Database System with Procedures*. ACM Trans. Database Syst., vol. 12, no. 3, pages 350–376, 1987. 29

[95] Michael Stonebraker and Greg Kemnitz. *The Postgres Next Generation Database Management System.* Commun. ACM, vol. 34, no. 10, pages 78–92, 1991. 29

[96] Michael Stonebraker, Jacek Becla, David Dewitt, Kian tat Lim and Stan Zdonik. *Requirements for Science Data Bases and SciDB.* In Proc. Conference on Innovative Data Systems Research (CIDR'09), 2009. 3

[97] Michael Stonebraker, Daniel Abadi, David J. DeWitt and Sam Madden. *MapReduce and Parallel DBMSs: Friends or Foes?* Commun, vol. 53, no. 1, pages 64–71, 2010. 3, 23

[98] *Sybase.* `http://www.sybase.com/products`. 67

[99] Alexander S. Szalay, Peter Kunszt, Ani Thakar, Jim Gray and Don Slutz. *Designing and Mining Multi-terabyte Astronomy Archives: the Sloan Digital Sky Survey.* In Proc. of the SIGMOD Conf., pages 451–462, 2000. 3

[100] R. Talmage. *Database Mirroring in SQL Server 2005.* Rapport technique, Microsoft, Apr 2005. 14

[101] Jiaqi Tan, Soila Kavulya, Rajeev Gandhi and Priya Narasimhan. *Visual, Log-Based Causal Tracing for Performance Debugging of MapReduce Systems.* In Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems, ICDCS '10, pages 795–806, 2010. 24

[102] Masahiro Tanaka and Osamu Tatebe. *Pwrake: A Parallel and Distributed Flexible Workflow Management Tool for Wide-area Data Intensive Computing.* In Proceedings of HPDC'10, pages 356–359, 2010. 27

[103] Kenjiro Taura, Takuya Matsuzaki, Makoto Miwa, Yoshikazu Kamoshida, Daisaku Yokoyama, Nan Dun, Takeshi Shibata, Choi Sung Jun and Jun'ichi Tsujii. *Design and Implementation of GXP Make – a Workflow System Based on Make.* In e-Science 2010 conference (eScience2010), 2010. xiii, 1, 16, 26, 27, 93

[104] Ian Taylor, Matthew Shields, Ian Wang and Andrew Harrison. *The Triana Workflow Environment: Architecture and Applications.* Workflows for e-Science, pages 320–339, 2007. 1, 17, 26, 27

[105] *Teradata: Data Appliance, Data Warehouse, Business Intelligence.* `www.teradata.com`. 2

[106] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff and Raghotham Murthy. *Hive - A Warehousing Solution Over a Map-Reduce Framework*. In Proceedings of VLDB Endow, pages 1626–1629, 2009. 2, 24, 30, 31, 90

[107] *TPC-H*. http://www.tpc.org/tpch. 36, 66

[108] Prasang Upadhyaya, YongChul Kwon and Magdalena Balazinska. *A Latency and Fault-tolerance Optimizer for Online Parallel Query Plans*. In SIGMOD Conference, pages 241–252, 2011. 31

[109] Patrick Valduriez. *Parallel Database Systems: Open Problems and New Issues*. Distributed and Parallel Databases, vol. 1, no. 2, pages 137–165, 1993. 13

[110] W.M.P. van der Aalst and A.H.M. ter Hofstede. *YAWL: Yet Another Workflow Language*. Inf. Syst., vol. 30, pages 245–275, 2005. 26

[111] *Vertica*. http://www.vertica.com/. 2, 23, 26, 30

[112] *VerticaInputFormat*. http://www.vertica.com/mapreduce. 26

[113] Rares Vernica, Michael J. Carey and Chen Li. *Efficient Parallel Set-similarity Joins Using MapReduce*. In SIGMOD Conference, pages 495–506, 2010. 25

[114] Jianwu Wang, Daniel Crawl and Ilkay Altintas. *Kepler + Hadoop: a General Architecture Facilitating Data-intensive Applications in Scientific Workflow Systems*. In SC-WORKS, 2009. 29

[115] Guozhang Wang, Marcos Vaz Salles, Benjamin Sowell, Xun Wang, Tuan Cao, Alan Demers, Johannes Gehrke and Walker White. *Behavioral Simulations in MapReduce*. Proc. VLDB Endow., vol. 3, no. 1-2, pages 952–963, September 2010. 25

[116] T. White. Hadoop: The definitive guide. O'Reilly Media, Inc., June 2009. 2, 23, 27, 28, 30, 31, 90, 94

[117] Yu Xu, Pekka Kostamaa and Like Gao. *Integrating Hadoop and Parallel DBMS*. In SIGMOD Conference, pages 969–974, 2010. 25

[118] Christopher Yang, Christine Yen, Ceryen Tan and Samuel Madden. *Osprey: Implementing MapReduce-style Fault Tolerance in a Shared-nothing Distributed Database*. In ICDE, pages 657–668, 2010. 26, 31

[119] Jia Yu and Rajkumar Buyya. *A Taxonomy of Scientific Workflow Systems for Grid Computing*. SIGMOD Record, vol. 34, no. 3, pages 44–49, 2005. 18

[120] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda and Jon Currey. *DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language*. In OSDI, pages 1–14, 2008. 25

[121] Li Yu, Christopher Moretti, Andrew Thrasher, Scott J. Emrich, Kenneth Judd and Douglas Thain. *Harnessing Parallelism in Multicore Clusters with the All-Pairs, Wavefront, and Makeflow Abstractions*. Cluster Computing, vol. 13, no. 3, pages 243–256, 2010. 26, 27

[122] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun and M. Wilde. *Swift: Fast, Reliable, Loosely Coupled Parallel Computation*. In IEEE International Workshop on Service Computing, pages 199–206, 2007. 1, 16, 26, 27

[123] Daniel Zinn, Sven Kohler, Shawn Bowers and Bertram Ludascher. *Parallelizing XML Processing Pipelines via MapReduce*. Rapport technique, 2009. 29

[124] M. Tamer Özsu and Patrick Valduriez. *Distributed and Parallel Database Systems*. ACM Computing Surveys, vol. 28, pages 125–128, 1996. 7