

InfiniteDB: a PC-Cluster based Parallel Massive Database Management System

Jianzhong Li
Harbin Institute of Technology,
China
lijz@hit.edu.cn

Hong Gao
Harbin Institute of Technology,
China
honggao@hit.edu.cn

Jizhou Luo
HIT Software Engineering
Corporation, Ltd, China
jizhou@mail.banner.com.cn

Shengfei Shi
HIT Software Engineering
Corporation, Ltd, China
sfshi@mail.banner.com.cn

Wei Zhang
HIT Software Engineering
Corporation, Ltd, China
zhangwei@mail.banner.com.cn

ABSTRACT

This paper describes a PC-cluster based parallel DBMS, InfiniteDB, developed by the authors. InfiniteDB aims at efficiently storing and processing of massive databases in response to the rapidly growing in database size and the need of high performance analyzing of massive databases. It supports the parallelisms of intra-query, inter-query, intra-operation, inter-operation and pipelining. It provides effective strategies for processing massive databases including the multiple data declustering methods, the declustering-aware algorithms for the execution of relational operations and other database operations, and the adaptive query optimization method. It also provides the functions of parallel data warehousing and data mining, the coordinator-wrapper mechanism to support the integration of heterogeneous information resources on the Internet, and the fault tolerant and resilient infrastructures. It has been used in many applications and has proved quite effective for storing and processing massive databases in practice.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems-Parallel databases

General Terms

Algorithms, Performance

Keywords

Parallel database, data declustering, parallel algorithm, parallel query processing

1. INTRODUCTION

Currently, there is a rapidly growing in database size and the need of high performance analyzing of massive databases.

Typical examples of massive databases include large scientific databases (e.g. the dataset of astronomical information, with the size of 3.65 terabyte(TB) per year, observed by Astronomical Observatory of China using LAMOST), large business transaction databases (e.g. the dataset of mobile phone call records, with the size of 300TB per year, collected by the China Net Corp.), archives of the medical images (e.g. the datasets of high resolution confocal light microscopy, CT imaging, MRI, and sonography, with the size of 60GB per day, collected by the First Affiliated Hospital of Harbin Medical University, China), and so on. Applications on these massive databases usually pose complex queries and require short response time. Over the past years, we worked with several research groups to understand the processing requirements for such massive databases. We have learned that a traditional (or serial) database management system (DBMS) cannot effectively handle so large databases.

The requirement of massive database management has fueled the need for parallel DBMS, which enables storing and processing of terabytes of data. In the past, a number of research prototypes, including Gamma [4], Bubba [3] and XPRS [14], tried to understand the issues in parallel databases. These and other projects addressed important issues such as parallel algorithms for the execution of database operations, query optimization techniques, data declustering and database system performance. The results of these studies form a basis for the knowledge of parallel database issues today. Through the research, several vendors shipped parallel DBMS products running on commercial parallel platforms such as IBM's DB2 PE [2]. However, all of the parallel database platforms were either SMP machines such as Sun's UltraEnterprise or proprietary hardware such as IBM's SP-2 and NCR's WorldMark, which are expensive so that the small and medium-sized enterprises in developing countries, such as China, cannot afford.

Looking over the recent parallel processing technology trends, PC-clusters, which are numbers of personal computers connected by various communication networks, are very promising platforms for storing and processing of massive databases. The great availability, scalability, flexibility and I/O parallelism can be achieved by PC-clusters. The low-cost of PC-clusters makes the parallel DBMSs running on PC-clusters (PC-cluster DBMSs for short) attractive to the enterprisers in developing countries such as China. Since the last decade of 20th century, a number of researchers have

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'07, June 12–14, 2007, Beijing, China.

Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...\$5.00.

focused on the PC-cluster DBMS. Recently, some commercial DBMS products that support for parallel processing of databases on PC-clusters have been shipped by vendors. In 1999, Oracle released the Oracle Parallel Server (*OPS*) [5]. *OPS* is a parallel DBMS with shared-disk architecture, and can be run on special PC-clusters with shared disks. *OPS* has the following disadvantages. First, since *OPS* requires that PC-clusters must have special hardware to support disk sharing, its application range is limited. Second, the shared-disk architecture restricts the I/O parallelism of PC-clusters. Third, the performance suffers significantly if the database partitioning is not carried out correctly since the cost of maintaining the coherent caches will increase. Finally, since the inter-PC-node synchronization issue is not well solved, the performance of *OPS* is low. In 2001, Oracle released the improved *OPS*, called Real Application Cluster (*RAC*), which integrated the advantages of Oracle 9i and the cache fusion technique to reduce the cost of inter-PC node synchronization but still has the drawbacks induced by the shared-disk architecture. To provide the support for enterprise grid computing, *RAC* is upgraded to *RAC 10g* and its two versions were released in 2004 and 2005. In 2004, IBM released the DB2 UDB Enterprise Server Edition (*ESE*) [6]. It has the database partitioning feature (*DPF*). With *DPF*, users can partition their databases across multiple servers or within a large *SMP* server and implement parallel processing of databases. *ESE* can be considered as a parallel DBMS with shared-nothing architecture. *ESE* can be ran on PC-clusters. However, since it is not developed purely for PC-clusters, it has some drawbacks when it is used for parallel processing on PC-clusters [1]. For example, it has only one data partition method, the hash-partition. The hash partition is not effective for the queries with join operations on non-partition attributes since it brings much inter-PC-node communication during the processing of the queries.

In order to efficiently store and process massive databases on PC-clusters, we developed a PC-cluster DBMS, called InfiniteDB, with shared-nothing architecture. All the characteristics of PC-clusters were taken into consideration during the developing of InfiniteDB, and many new strategies were adopted to maximize the availability, scalability, flexibility and the performance of InfiniteDB. In contrast to other parallel DBMSs, InfiniteDB has the following distinguishing features.

First, it supports two levels of parallelisms. The first level parallelism is intra-query parallelism and inter-query parallelism. The second level parallelism concerns the parallel execution of the operations in a query, including intra-operation parallelism, inter-operation parallelism and pipelining parallelism.

Second, it provides a number of strategies for high performance processing of massive databases, including multiple data declustering methods among PC-nodes to minimize the inter-PC-node communication and maximize the parallelism of PC-clusters during query processing, a set of declustering-aware algorithms for the execution of relational operations and other database operations to take the advantages of the underlying data declustering methods during query processing, and a self-adaptive and resource-aware query optimization method.

Third, it moves the data warehousing and data mining from tools into the backend and provides the functions of parallel data warehousing and parallel data mining in the

kernel of InfiniteDB to efficiently support business intelligence applications.

Fourth, it provides a coordinator-wrapper mechanism with a global data model and a global query language to support the integration of heterogeneous information resources on the Internet.

Fifth, besides the traditional database recovery subsystem, it provides two levels of fault tolerant and resilient infrastructures. The one at the first level is the intra-PC-node fault tolerant and resilient infrastructure that automatically places redundant copies of the data in a PC-node to multiple disks connected to the PC-node and automatically maintains the consistency of the redundant copies. The one at the second level is the inter-PC-node fault tolerant and resilient infrastructure. It allows users to place redundant copies of any dataset in a PC-cluster to any group of PC-nodes in the PC-cluster to ensure that the data will be protected against the failure of any PC-node. The consistency of the redundant copies is maintained automatically.

InfiniteDB is released in 2001, and has been used by many government organizations and enterprises in China. For example, it has been used for 6 years by local taxation bureaus of 2 provinces and 23 cities in China, and has won the National Science and Technology Progress Award (II) from the central government of China in 2004.

This paper focuses on the pivotal aspect of InfiniteDB, i.e. the mechanism of storing and parallel processing of massive databases, including the data declustering subsystem, which has significant impact on parallel query processing, parallel algorithms package for the execution of database operations, in which the declustering-aware parallel algorithms are included, and the parallel query optimization and processing subsystem. The applications of InfiniteDB are also presented. The rest of the paper is organized as follows. Section 2 presents the data declustering subsystem. Section 3 describes the parallel algorithms package, especially the declustering-aware parallel algorithms. Section 4 summarizes some key features of the query optimization and processing subsystem. In section 5, the applications of InfiniteDB are discussed. We conclude with some practical lessons in section 6.

2. DATA DECLUSTERING SUBSYSTEM

Data declustering is to partition a relation into subsets and distribute them among the PC-nodes in a server (a PC-cluster). Data declustering has significant impact on the performance of parallel query processing. To minimize the time cost of query processing, the data declustering subsystem of InfiniteDB supports four kinds of data declustering methods, i.e. 1-dimensional declustering, multidimensional declustering, cloning, and parallel B-trees. All the methods decluster a relation among all or some of the PC-nodes of a PC-cluster. In the following discussion, let $\{N_0, N_1, \dots, N_{n-1}\}$ be the set of n PC-nodes among which the data will be declustered and $R(A_1, A_2, \dots, A_k)$ be a relation with k attributes $A_1, A_2, \dots$, and A_k .

2.1 1-Dimensional Data Declustering Methods

The basic idea of 1-dimensional data declustering is to partition a relation according to the value of one attribute of the relation. The attribute is called declustering attribute. InfiniteDB supports three 1-dimensional data declustering methods. They are the round-robin declustering method,

the hash declustering method and the range declustering method.

2.1.1 Round-robin declustering method

Let t_i be the i^{th} tuple of relation R . By Round-Robin declustering method, relation R will be distributed among n PC-nodes as following: t_i is allocated to the $(i \bmod n)^{th}$ PC-node in $\{N_0, N_1, \dots, N_{n-1}\}$. Round-Robin declustering method is effective for the queries that access a relation by sequentially scanning the whole relation. The problem with the method is that it cannot efficiently support the queries that associatively access tuples. This is because that this kind of queries only accesses few tuples but all the PC-nodes in $\{N_0, N_1, \dots, N_{n-1}\}$ must be in action.

2.1.2 Hash declustering method

Let A be the declustering attribute of relation R and V be the domain of A . In hash declustering method, a hash function $H: V \rightarrow \{0, 1, \dots, n-1\}$ needs to be defined. Let t be a tuple of R . By hash declustering method, t is allocated to the PC-node $H(t[A])$ in $\{N_0, N_1, \dots, N_{n-1}\}$, where $t[A]$ is the value of t on attribute A . Hash declustering method is ideally suited for queries that sequentially and associatively access to data. Associative access to the tuples with a specific value of the declustering attribute can be directed to a single PC-node, avoiding the overhead of starting queries on multiple PC-nodes. The problem with the method is that it risks the data skew in which most of data are placed in a small number of PC-nodes and the execution skew in which all the executions occur in a small number of PC-nodes.

2.1.3 Range declustering method

Range declustering method first divides the domain of the declustering attribute A into n ranges $I_0, I_1, \dots, I_{n-1}$. Then, relation R is partitioned into n subsets $S_0, S_1, \dots, S_{n-1}$, where $S_i = \{t | t \in R \text{ and } t[A] \in I_i\}$. Finally, S_i is allocated to the i^{th} PC-node in $\{N_0, N_1, \dots, N_{n-1}\}$ for i from 1 to $n-1$. Range declustering method is not only good for sequential and associative access, but also good for clustering or range access. Similar to the hash declustering method, the method also risks data skew and execution skew.

2.2 Multidimensional Data Declustering

All one-dimensional declustering methods partition a relation using the values of a single attribute so that operations applied to the declustering attribute can be performed very efficiently. However, these methods would not be able to efficiently support the queries that involve non-declustering attributes. To solve this problem, InfiniteDB provides a multidimensional declustering method, called coordinate modular declustering (CMD for short) [13].

In CMD method, a relation with k attributes is treated as a subset of a k -dimensional space. Given a relation $R(A_1, A_2, \dots, A_k)$, there must be a k -dimensional space, of which R is a subset. The CMD method partitions the k -dimensional space into subspaces by dividing the domain of each dimension into intervals so that all subspaces contain the same or near the same number of tuples in R . Thus, R is also partitioned into n subsets. Each subspace is uniquely allocated to a PC-node in $\{N_0, N_1, \dots, N_{n-1}\}$ by the coordinate modular function $CMD(x_1, \dots, x_k) = (x_1 + \dots + x_k) \bmod n$, where $(x_1, \dots, x_k)$ is the coordinate of the subspace. The subset of

R in the subspace $(x_1, \dots, x_k)$ is allocated to the PC-node numbered by $CMD(x_1, \dots, x_k)$. By a mapping, the subspace on each PC-node is compressed into a k -dimensional hyper-rectangle such that the grid file, which is very efficient for range queries, can be used to organize it.

The CMD method has the following advantages.

- (1). Since data declustering is based on all dimensions, queries with predicate defined on any attributes (dimensions) can be performed efficiently.
- (2). The method is balanced for relations with stationary data distribution, and thus expensive data rebalancing is usually not needed. Experimental results show that the method works well even for some databases without stationary data distributions.
- (3). Since the method is balanced in terms of data volume on each PC-node and neighboring regions in multidimensional space are on different PC-nodes, maximum parallelism can be achieved during query processing.
- (4). The method is optimal for range queries in term of the I/O workload balance due to the balance of the method mentioned in 3.
- (5). The grid block split and merge operations are localized to a single PC-node so that data transmission among PC-nodes during data maintenance is avoided. Thus, the method reduces the cost of insertion, deletion and update.

2.3 Parallel B-trees

B-tree is very efficient for organizing databases on disks in practice. Apparently, it is meaningful to apply B-tree to a parallel database. Thus, InfiniteDB provides the parallel B-tree indices [7] including the parallel B-tree with record distribution (*RDB-tree*), parallel B-tree with large page (*LPB-tree*) and parallel B-tree with node distribution (*NDB-tree*).

The basic idea underlying the *RDB-tree* is to use multiple B-trees, one for each PC-node, and distribute the records among these B-trees. Many methods for allocating records to PC-nodes are feasible, such as any one of previous 1-dimensional declustering methods, giving rise to different variants of the basic *RDB-tree* approach. Typically, records do not migrate between PC-nodes. Once a record has been allocated to a PC-node, it remains on that PC-node until deleted.

The basic idea of the *LPB-tree* is to use very large page, called super page, for each node of B-tree, and distribute the data in each super page across $\{N_0, N_1, \dots, N_{n-1}\}$ using any one of previous 1-dimensional declustering methods. A super page thus consists of n subpages, each stored on a different PC-node. Retrieval of a super page requires n reads, one read per PC-node.

The *NDB-tree* is based on the idea of using an ordinary B-tree but distributing the nodes of the B-tree across $\{N_0, N_1, \dots, N_{n-1}\}$. This requires that each index entry of B-tree is expanded to indicate which PC-node the pointer refers to. There are many methods to distribute the nodes of the B-tree. One method is random distributing, i.e. when a new node of B-tree is created it is randomly allocated to one of the n PC-nodes.

2.4 Cloning

For efficiently processing the queries with join operations, we must carefully choose the declustering attribute to allow for co-located join, i.e. a join that can be performed without the inter-PC-node communication. Using the hash

declustering method, we can easily make a join co-located by letting the declustering attributes be the join attributes. However, it is not possible to co-locate all the joins in some schemas. To solve this problem, InfiniteDB provides a cloning method that allows users to create replicated copies of a relation on all PC-nodes in $\{N_0, N_1, \dots, N_{n-1}\}$. The consistency of the replicated copies is maintained automatically by the data declustering subsystem of InfiniteDB.

3. PARALLEL ALGORITHM PACKAGE AND DECLUSTERING-AWARE ALGORITHMS

The parallel algorithm package of InfiniteDB provides all the parallel algorithms for the execution of the relational operations, other database operations, data warehousing operations and data mining operations. It has been shown that if a database operation algorithm is aware of the underlying data declustering methods of the operand relations, it can take the advantages of the data declustering methods to reduce the cost of the algorithm and to improve resource utilization [9, 10]. To be aware of the declustering methods, the parallel algorithm package provides a set of declustering-aware parallel algorithms for the execution of each database operation, one algorithm per declustering method. The following subsections concentrate on the declustering-aware parallel algorithms in the package.

3.1 Overview of Declustering-Aware Algorithms

There are five classes of declustering-aware parallel algorithms in the parallel algorithm package of InfiniteDB. The first class is for the relational operations, such as select, join, project, order by, group by, and set operations. The second class is for the data maintenance operation, such as insert, update and delete. The third class is for the database definition operation, such as create table and index. The fourth class is for the data partition maintenance operations, such as redeclustering, data load and unload. The fifth class is for the data warehousing and data mining operations, such as cube, aggregation, recursive and iterative operations.

For example, for the join operation, there are three CMD-declustering oriented parallel algorithms, six B-trees oriented parallel algorithms, one range-declustering oriented parallel algorithm, and one hash-declustering oriented parallel algorithm in the first class of declustering-aware parallel algorithms.

During the query optimization, the best suitable algorithm for each operation in a query can be chosen to take the advantages of the underlying data declustering methods and reduce the cost of query processing. From applications, it is shown that the declustering-aware parallel algorithms outperform the algorithms without taking the underlying data declustering methods into consideration.

To show the principle of declustering-aware algorithms, we illustrate one of the *RDB-tree* oriented parallel join algorithms, *RDB-Tree-Join*, and one of the *CMD-clustering* oriented parallel join algorithms, *CMD-Join* algorithm, in the following.

3.2 RDB-Tree-Join Algorithm

RDB-Tree-Join Algorithm is suitable for the case of the join relations having the indices of *RDB-tree* with range-

declustering strategy on their join attributes. In the following discussion, let R and S be the operands of a join operation, both R and S have the *RDB-tree* indices on their join attributes, the size of R is greater than the size of S , P is the number of available PC-nodes.

Let the domain of the index attribute of R be divided into P intervals, $I_1 = [a_0, a_1)$, $I_2 = [a_1, a_2)$, $\dots$, $I_{P-1} = [a_{P-2}, a_{P-1})$ and $I_P = [a_{P-1}, a_P]$, and the domain of the index attribute of S be divided into P intervals, $J_1 = [b_0, b_1)$, $J_2 = [b_1, b_2)$, $\dots$, $J_{P-1} = [b_{P-2}, b_{P-1})$ and $J_P = [b_{P-1}, b_P]$. Unless there is confusion, we also use $[x_{k-1}, x_k)$ (or $[x_{k-1}, x_k]$) to express the subset, $\{t | t \in X, t[A] \in [x_{k-1}, x_k) \text{ (or } [x_{k-1}, x_k])\}$, of relation X , where $t[A]$ is the value of tuple t on the index attribute A . In this case, we simply call $[x_{k-1}, x_k)$ (or $[x_{k-1}, x_k]$) an interval of X . R and S can be expressed by the intervals $[a_0, a_P]$ and $[b_0, b_P]$ respectively. The intervals I_k and J_k are in the k^{th} PC-node for $1 \leq k \leq P$.

The idea of the *RDB-Tree-Join* algorithm is to redistribute the tuples of S among the processing nodes according to the distribution of R first, and then generate the join result of R and S by joining the subsets of R and S in each PC-node in parallel. Since S is the smaller relation, lower communication is incurred in distributing the tuples of S among the PC-nodes while the tuples of R are fixed. The *RDB-Tree-Join* algorithm has three phases, namely boundary processing phase, interval tuning phase, and join phase.

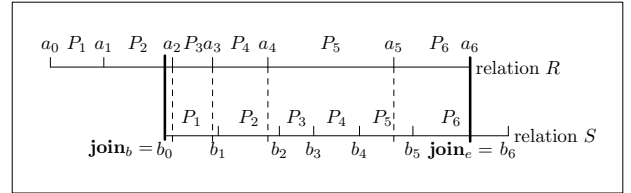


Figure 1: The join interval of relations R and S

The boundary processing phase is to determine the join interval of R and S and the strategy for redistributing S among the processing nodes. In this phase, the algorithm takes the characteristics of *RDB-tree* into consideration.

The join interval of R and S is easy to determine, which is $[Join_b, Join_e]$, where $Join_b = \max\{a_0, b_0\}$ and $Join_e = \min\{a_P, b_P\}$. An example of join interval is shown in figure 1, where $Join_b = b_0$ and $Join_e = a_6$. In figure 1, the label P_i on an interval denotes that the interval is stored in the PC-node P_i . It is clear that we do not need to consider the tuples of R and S beyond the join interval in performing the join of R and S .

While the join interval is easy to determine, it is difficult to determine the strategy for redistributing S . We first discuss how to compute the subset of S that needs to be transmitted among the PC-nodes.

DEFINITION 3.1. (Dislocated tuples). Let $[Join_b, Join_e]$ be the join interval of R and S , $[a_{i-1}, a_i]$ be an interval of R that is in the PC-node P_i , and $[a_{i-1}, a_i] \cap [Join_b, Join_e]$ be not empty. The tuples of S that are in $[a_{i-1}, a_i]$ but are not in the PC-node P_i are defined as the dislocated tuples of S related to $[a_{i-1}, a_i]$.

Since the interval $[a_{i-1}, a_i]$ of R is in the PC-node P_i , all the dislocated tuples of S related to $[a_{i-1}, a_i]$ must be transmitted to the PC-node P_i so that the tuples of R in $[a_{i-1}, a_i]$ can be joined with all the related tuples of S correctly in

the PC-node P_i . All the dislocated tuples of S related to $[a_{i-1}, a_i]$ can be expressed as an union of some intervals of S . For example, in figure 1, the dislocated tuples of S related to $[a_4, a_5]$ are in the interval $[a_4, b_2] \cup [b_2, b_3] \cup [b_3, b_4]$.

DEFINITION 3.2. (Dislocated-tuple set). *Each interval of S contained in the set of the dislocated tuples of S related to $[a_{i-1}, a_i]$ is called a dislocated-tuple set of S related to $[a_{i-1}, a_i]$.*

DEFINITION 3.3. (Redistribution subset). *The union of all the dislocated-tuple sets of S related to all the intervals of R is defined as the redistribution subset of S .*

For example, in Figure 1, the redistribution subset of S is the union of the dislocated-tuple sets $[b_0, a_2]$, $[a_2, a_3]$, $[a_3, b_1]$, $[b_1, a_4]$, $[a_4, b_2]$, $[b_2, b_3]$, $[b_3, b_4]$, and $[a_5, b_5]$. Next, we discuss how to determine the strategy for transmitting the tuples in the redistribution subset of S .

DEFINITION 3.4. (Strategy for transmitting dislocated-tuple set). *A strategy for transmitting a dislocated-tuple set $[c, d]$ of S related to $[a_{i-1}, a_i]$ is $([c, d], i, s_n, flag)$, which means that the dislocated-tuple set $[c, d]$, s_n and $flag$ are transmitted to PC-node P_i , where s_n is the order number of $[c, d]$ relative to all the dislocated-tuple sets of S related to $[a_{i-1}, a_i]$, and $flag$ indicates that PC-node P_i should begin to join R and S after it receives $[c, d]$ if $flag = 1$ or should not if $flag = 0$.*

DEFINITION 3.5. (Strategy for Transmitting Redistribution-subset). *A strategy for transmitting redistribution subset of S is defined as a set of $\{([c, d], i, s_n, flag) | [c, d] \text{ is a dislocated-tuple set of } S\}$.*

For example, the strategy for transmitting the redistribution-subset of S in figure 1 is

$$\begin{aligned} & \{([b_0, a_2], 2, 1, 1), ([a_2, a_3], 3, 1, 1), ([a_3, b_1], 4, 1, 0), \\ & \{([b_1, a_4], 4, 2, 1), ([a_4, b_2], 5, 1, 0), ([b_2, b_3], 5, 2, 0), \\ & \{([b_3, b_4], 5, 3, 1), ([a_5, b_5], 6, 1, 1)\}. \end{aligned}$$

In summary, the boundary processing phase consists of the following five steps.

Step 1. Determine the join interval $[Join_b, Join_e]$.

Step 2. Determine the redistribution subset of S .

Step 3. Determine the strategy $\{([c, d], i, s_n, flag) | [c, d] \text{ is a dislocated-tuple set of } S\}$.

Step 4. Partition the strategy $\{([c, d], i, s_n, flag) | [c, d] \text{ is a dislocated-tuple set of } S\}$ into the following P substrategies:

$$strategy_1 = \{([c_1, d_1], i_1, s_n, flag) | [c_1, d_1] \text{ is in } P_1\}$$

.....

$$strategy_P = \{([c_P, d_P], i_P, s_n, flag) | [c_P, d_P] \text{ is in } P_P\}.$$

Step 5. Send $strategy_i$ to PC-node P_i for $1 \leq i \leq P$.

In the interval tuning phase, P_i transmits the dislocated tuples of S to other PC-nodes according to $strategy_i$ in parallel for $1 \leq i \leq P$.

In the join phase, all the PC-nodes perform the join of R and S in parallel. Let R_i and S_i be the subsets of R and S to be joined in PC-node P_i . R_i is the set of tuples in the leaf nodes of the B-tree of R in P_i . R_i is a sorted set. S_i is the merge of the set of tuples in the leaf nodes of the B-tree of S in P_i , which is sorted, with the sorted subsets of the dislocated tuples of S sent by other PC-nodes. S_i is also a sorted set. Thus, the join performed by P_i is a merge of R_i and S_i for $1 \leq i \leq P$.

The main advantages of the RDB-Tree-Join algorithm are as follows.

(1). The characteristics of the parallel B-trees are adequately exploited in processing of the join operations.

(2). The problem size is pruned by not considering the portions of the operand relations that are irrelative to the result of the join operation.

Theoretical analysis shows that the parallel execution times of the RDB-Tree-Join algorithms are linearly proportional to $\max\{N/P, M/P\}$, where N and M are the sizes of the operand relations and P is the number of PC-nodes.

The RDB-Tree-Join algorithm has been compared to the well known parallel join algorithms, parallel hybrid-hash join, parallel Grace-hash join and parallel merge-sort join in practice. Experimental results show that the RDB-Tree-Join algorithm is more efficient than all other parallel join algorithms in case of the operand relations having *RDB-tree* indices on their join attributes.

3.3 CMD-Join Algorithm

In the following discussion, we call a relation stored by the CMD-declustering method as a CMD relation.

CMD-Join algorithm takes two CMD relations as input, and makes use of the following two advantages of the CMD relations.

(1). All CMD relations are partially sorted on all dimensions, i.e. if R is a CMD relation, I_{k_i} and I_{k_j} are two intervals on the k^{th} dimension with corresponding attribute A , and $i < j$, then for $\forall u \in \{t | t \in R, t[A] \in I_{k_i}\}$ and $v \in \{t | t \in R, t[A] \in I_{k_j}\}$, $u[A] < v[A]$.

(2). All CMD relations are declustered by partitioning all the domains of all attributes so that the joins of CMD relations with any attributes as join attributes can be efficient supported.

Given CMD relations R and S with n and m dimensions or attributes, CMD-Join performs the join of R and S in two phases.

In phase 1, CMD-Join computes the set of related n -dimensional regions of R that need to be joined with S and the set of related m -dimensional regions of S that need to be joined with R using the feature of CMD relation being partially sorted on all dimensions.

In phase 2, CMD-Join performs the join of R and S by joining the related regions of R with the related regions of S using the hash method.

The analysis and experimental results show that CMD-Join performs much better than other parallel join algorithms for CMD relations. It has the following advantages as well.

(1). It has the ability to avoid reading parts of relations that need not be joined.

(2). It requires less memory than the others parallel join algorithms and is disk-read optimal.

(3). It can achieve workload balance without the necessity of an expensive rebalancing.

(4). It works better than other parallel join algorithms for conditions of high data skew.

(5). A combination of select and join (where the select is a range query) can be accomplished using any set of attributes without writing intermediate files to disk in more situations than other parallel join algorithm.

4. QUERY OPTIMIZATION AND PROCESSING SUBSYSTEM

The query optimization and processing subsystem of InfiniteDB is specially developed for efficiently processing of

massive databases and aimed at minimizing the time cost of query processing. In this section, we first describe the architecture for query processing, and then focus on the most important part of the query optimization and processing subsystem, i.e. query optimizer.

4.1 Architecture for Query Processing

The query optimization and processing subsystem (*QOPS*) has the client-server architecture. A PC-cluster serves as the server and some other front-end machines connected to the PC-cluster can be the clients.

Figure 2 depicts the architecture of *QOPS*, which is not the whole architecture of InfiniteDB. *QOPS* consists of four components, the query optimizer, the parallel operation algorithm package, the coordinator and the query executor. The query optimizer and the coordinator run at the clients. The query executor and the parallel operation algorithm package run at the server (a PC-cluster).

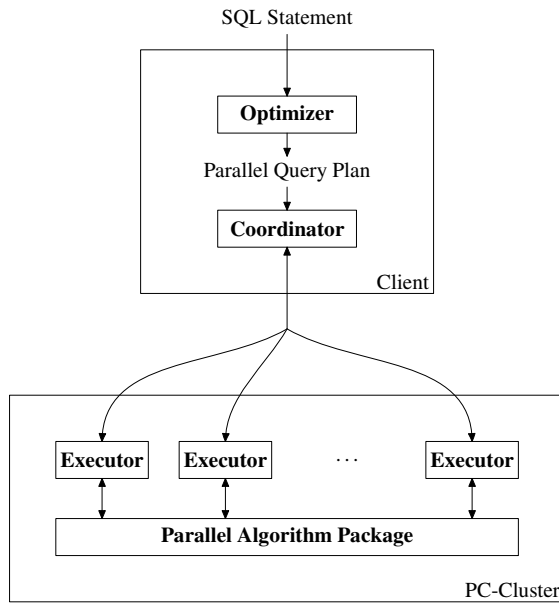


Figure 2: The architecture of *QOPS*

Queries issued from users or applications are optimized by the query optimizer at clients to produce the parallel query plans. The plans are passed to the coordinators at the clients.

Execution of a query is initiated by a coordinator. The coordinator coordinates related PC-nodes of the server to execute the generated parallel plan, collects the results from the PC-nodes, generates the final answer of the query, and passes it to the users or applications.

Each PC-node of the server executes a query executor. The query executor is a process that consists of several permanent threads, and one of such threads, called interface thread, interacts with the coordinator. It creates query processing threads for executing the query processing tasks assigned by the coordinator, and monitors the query processing threads and the coordinator. After the query processing threads finish, it reports the result to the coordinator and cancels the query processing threads.

While the query plan is being executed, the parallel algorithms for operations such as select and join need to be

invoked. These parallel algorithms are contained in the parallel algorithm package of InfiniteDB, and are dynamically revoked during query processing.

4.2 Query Optimizer

The key part of the query optimization and processing subsystem of InfiniteDB is the query optimization method that aims at finding a parallel plan that delivers the query result in minimal time. While considerable progress has been made in query optimization, many query optimization problems in parallel database systems still remain open. To counter these problems, InfiniteDB provides a multi-weighted tree based query optimizer [11, 8]. It depends on a multi-weighted tree based parallel query plan model and a cost model for the parallel query plans. It has the following distinguished features.

(1) It uses the multi-weighted tree to model parallel query plans so that all aspects of a parallel query plan can be easily expressed.

(2) A mathematical model of waiting time of the operations in pipelining execution on PC-clusters is developed, and the cost model of query plans takes the waiting time into consideration.

(3) The optimizer minimizes the waiting time of the operations in pipelining execution that has significant impact on the time cost of the query processing in PC-cluster DBMSs.

(4) The optimizer is aware of the memory resource, the computing resource, the declustering methods and the multiple parallel algorithms for each database operation,

(5) it is adaptive for queries with different complexities.

In the rest of the subsection, we first describe the model of the parallel query plans, then describe how to compute the time cost of the parallel query plans, and finally present the query optimizer.

4.2.1 Model of Parallel Query Plans

The parallel query plan model of InfiniteDB is the first one to model all basic relational operations, all three types of parallelism of a query execution, i.e. intra-operator, inter-operator and pipelining parallelisms, the allocation of PC-nodes and memory for operations, the allocation of algorithms for operations, the allocation of memory for the buffers between operations in pipelining execution, and the data re-distribution among PC-nodes.

In the model of parallel query plans, the following atomic sub-operations are used to express the algorithms for implementing relational operations: (1) *scan* that scans the entire relation to implement *non-indexed select*, *project*, *re-distribute*, etc, (2) *sort* that sorts the input, (3) *hash* that builds a hash table from the input, (4) *probe* that uses input to probe hash table and output hits, (5) *merge* that merges two sorted inputs, (6) *nested-loops* that compares each tuple of one relation against each tuple of the other relation, (7) *CMD-join_i* that joins input CMD relations by the i^{th} CMD-join algorithm [9], and (8) *B-tree-join_i* that joins input relations by the i^{th} parallel B-tree-join algorithm [10].

DEFINITION 4.1. Let $G = (V, E)$ be a tree, and $S_1, \dots, S_m, T_1, \dots, T_k$ be sets. A multi-weighted tree is $ET = (G, NF, EF)$, where NF is a function $V' \rightarrow (S_1 \times \dots \times S_m)$, and EF is a function $E' \rightarrow (T_1 \times \dots \times T_k)$, where $V' \subseteq V$ and $E' \subseteq E$.

DEFINITION 4.2. Let Q be a query, N be the number of PC-nodes, M be the memory size in bytes, OP be the set of

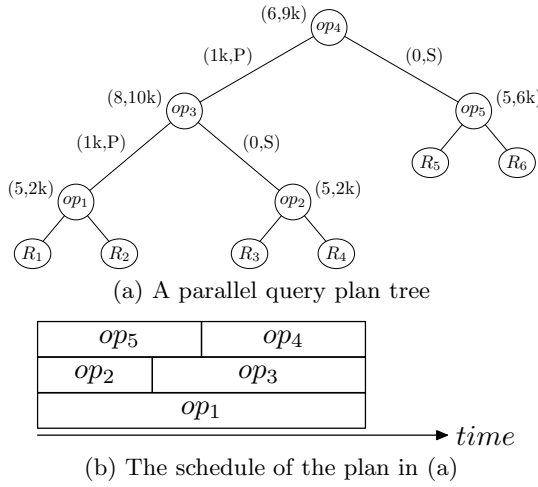


Figure 3: A parallel query plan and its schedule

atomic operations in Q , and φ be the set of operand relations in Q . A parallel query plan of Q is a multi-weighted tree, $((V, E), F_1, F_2)$, where $V = OP \cup \varphi$, $E = \{(X, Y) | (X, Y \in OP) \text{ or } (Y \in OP \text{ and } X \in \varphi)\}$, F_1 is the function $OP \rightarrow \{0, 1, \dots, N\} \times \{0, 1, \dots, M\}$, F_2 is the function $E \rightarrow \{(X, Y) | (Y \in OP \text{ and } X \in \varphi)\} \rightarrow \{0, 1, \dots, M\} \times \{P, S\}$. The parallel query plan space of Q is the set of all the parallel query plans of Q .

It is obvious that the allocation of PC-nodes and memory for operators is modeled by function F_1 . F_1 also models the intra-operation parallelism by assigning multiple PC-nodes to an operation. The pipelining parallelism is modeled by function F_2 . If F_2 puts weight (m, P) on edge (op_1, op_2) , then the operators op_1 and op_2 are executable in pipelining and the buffer size assigned to the pipeline is m . Otherwise, the weight $(0, S)$ on edge (op_1, op_2) forces op_1 and op_2 to be executed sequentially. From the definition of the edge set E , all independent operations must be on different paths of the parallel query plan tree, which models the inter-operation parallelism. figure 3 shows a parallel query plan tree and the schedule for parallel execution of the query plan, where op_i s are atomic operations, R_i s are relations, the label (x, y) on each operation node is the value of function F_1 on the node, and the label (m, z) on each edge is the value of function F_2 on the edge. The query plan tree is equal to the expression $((R_1 op_1 R_2) op_3 (R_3 op_2 R_4)) op_4 (R_5 op_5 R_6)$,

4.2.2 Time Cost of Parallel Query Plans

To compute the cost of the multi-weighted tree based parallel query plans, we developed two mathematical models. One is the response time of individual operations. It is modeled as a multi-variable function $rtime(op, A, n, m, \rho, \theta)$, where op is an operation, A is the algorithm for executing op , n is the number of PC-nodes assigned to op , m is the memory size assigned to op , ρ is the set of sizes of the operand relations, and θ is the size of the result.

The other mathematical model is the waiting time of the operators in pipelining execution. Let op_y and op_x be two operations in pipelining execution, op_y be the producer, and op_x be the consumer. There must be a pipelining buffer holding m tuples between op_x and op_y . op_y produces tuples to the buffer and op_x retrieves tuples from the buffer. The

buffer may overflow or underflow. In case of underflowing, op_x has to wait. In case of overflowing, op_y has to wait. The waiting time has significant impact on the cost of parallel query plans. We developed a mathematical model of the waiting time [8]. For any op_x in a pipeline, the waiting time of op_x is represented by WT_x in the following discussion.

Besides $rtime(op, A, n, m, \rho, \theta)$ and WT_x , we also need the concept of elapsed time of an operation op in a plan. It is the elapsed time from the starting time of the plan to the termination of op , denoted by $etime(op)$.

The time cost of an entire query plan is defined as the elapsed time from the starting time of the plan to the termination time of the plan. Let P be a parallel query plan with k operations, $\alpha = \{a_1, \dots, a_k\}$, $\eta = \{n_1, \dots, n_k\}$, $\sigma = \{m_1, \dots, m_k\}$, $\rho = \{s_1, \dots, s_t\}$, and $\theta = \{o_1, \dots, o_k\}$, where a_i , n_i and m_i are the algorithm, number of PC-nodes and memory size allocated for the i^{th} operation in P , s_i is the set of the operand relations' sizes, and o_i is the size of the result of the i^{th} operation in P . The time cost of a parallel query plan P can be recursively computed as following:

1. For each operation node op_x directly connected to leaf nodes in P , $etime(op_x) = rtime(op_x, a_x, n_x, m_x, s_x, o_x) + WT_x$.
2. For any other operation op_x in P , $etime(op_x)$ is computed as following.

Case 1. op_x has a son op_y , and the weight on edge (op_y, op_x) is $(0, S)$, i.e., op_x and op_y are sequentially executed. The subtree and its schedule are shown in figure 4. In the case, $etime(op_x) = rtime(op_x, a_x, n_x, m_x, s_x, o_x) + etime(op_y) + WT_x$.

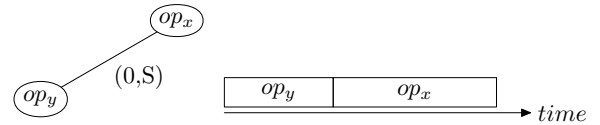


Figure 4: A subtree of P and its schedule

Case 2. op_x has a son op_y and the weight on edge (op_y, op_x) is (m, P) , i.e. op_x and op_y are executed in pipeline. The subtree and its schedule are shown in figure 5. In this case, $etime(op_x) = (etime(op_y) - rtime(op_y) - WT_y) + WT_x + rtime(op_x, a_x, n_x, m_x, s_x, o_x)$, where $(etime(op_y) - rtime(op_y) - WT_y)$ is the start time of op_x .

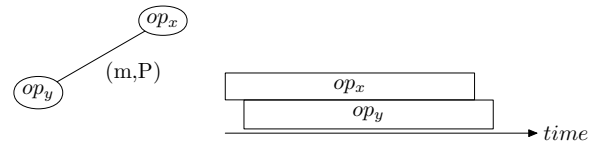


Figure 5: A subtree of P and its schedule

Case 3. op_x has two sons op_y and op_z , and the weights on edges (op_y, op_x) and (op_z, op_x) are $(0, S)$ and $(0, S)$, i.e., the execution of op_x and op_y is sequential, the execution of op_x and op_z

is sequential, and op_y and op_z are executed in parallel. The subtree and its schedule are shown in figure 6. In this case, $etime(op_x) = rtime(op_x, a_x, n_x, m_x, s_x, o_x) + WT_x + \max\{etime(op_y), etime(op_z)\}$.

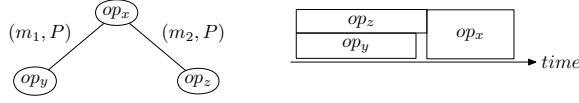


Figure 6: A subtree of P and its schedule

Case 4. op_x has two sons op_y and op_z , and the weights on edges (op_y, op_x) and (op_z, op_x) are (m_1, P) and (m_2, P) , i.e., op_x and op_y are executed in pipeline, op_x and op_z are also executed in pipeline, and op_y and op_z are executed in parallel. The subtree and its schedule are shown in figure 7. In this case, $etime(op_x) = rtime(op_x, a_x, n_x, m_x, s_x, o_x) + WT_x + \min\{etime(op_y) - rtime(op_y) - WT_y, etime(op_z) - rtime(op_z) - WT_z\}$.

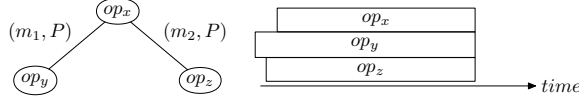


Figure 7: A subtree of P and its schedule

Case 5. op_x has two sons op_y and op_z , and the weights on edges (op_y, op_x) and (op_z, op_x) are $(0, S)$ and (m, P) , i.e., op_x and op_y are executed sequentially, op_x and op_z are executed in pipeline, and op_y and op_z are executed in parallel. The subtree and its schedule are shown in figure 8. In this case, $etime(op_x) = rtime(op_x, a_x, n_x, m_x, s_x, o_x) + WT_x + \max\{etime(op_y), etime(op_z) - rtime(op_z) - WT_z\}$.

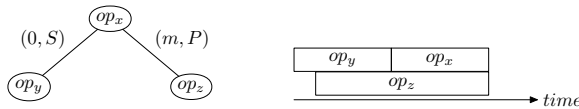


Figure 8: A subtree of P and its schedule

Case 6. op_x has two sons op_y and op_z , and the weights on edges (op_y, op_x) and (op_z, op_x) are (m, P) and $(0, S)$. This case is similar to case 5.

- Finally, the time cost of P is $etime(op_{root})$, where op_{root} is the root of P .

4.2.3 Query Optimizer

The query optimizer (QO) is self-adaptive. In case of the number of joins in a given query, Q , is smaller than 10, QO enumerates all the parallel plans of Q , computes the time cost of each plan using the method in subsection 4.2.2, and finally chooses the plan with the minimum cost to execute.

The enumeration algorithm can generate the optimal parallel plan and the cost of the algorithm is acceptable for the

queries with less than 10 joins. From our experience in practice, the number of joins in most of queries is smaller than 10. Thus, the enumeration algorithm is significant.

In case of the number of the joins in the query Q is greater than or equal to 10, QO uses a heuristic optimization method to choose a near-optimal parallel plan of Q to execute. The heuristic method consists of 2-phases.

In phase 1, Q is translated into a procedural plan or query tree and transformations are applied to obtain the plan with the heuristically minimal work (i.e. sequential execution time). Since it is what sequential optimizers do, QO uses a standard sequential optimizer in this phase.

In phase 2, the plan tree of Q is parallelized and the near-optimal parallel query plan, expressed by multi-weighted tree, is derived in the following six steps.

Step 1. For each operation in the query tree, the best parallel algorithm for implementing the operation is chosen by a heuristic method. The choice of the parallel join algorithms depends on many factors, such as the declustering method of the operand relations, the type of join (equal join or not), size of join relations, and so on.

Step 2. The query tree is transformed into operator tree T by expanding each node in the query plan into atomic operations. For example, a merge-sort join of relations R and S is decomposed into the atomic operation sequence of *sorting* S , *sorting* R and *merging* the results of the two sort operations.

Step 3. Determine pipelines in T . First, the weight (m, x) on each edge (w, v) of T is determined, where v is the parent of w . x is determined by heuristic rules. If $x = S$, $m = 0$. If $x = P$, m is computed using a mathematical model [8] to minimize the waiting time of v and w when they are in pipelining execution. Then, the natural pipelines of T is obtained by removing the edges with weight $(0, S)$ from T . Due to the limitation of the memory, all the operations on a pipeline may not be executed in parallel. If the total size of all pipeline buffers on a pipeline is greater than the size of available memory, the pipeline is partitioned into multiple sub-pipelines. The resulting pipelines are the final *executable pipelines*.

Step 4. Determine the strategies of executing pipelines in T and generate an *execution strategy graph*. A node of the *execution strategy graph* is a set of pipelines that can be executed in parallel, i.e. all the pipelines contained in the node are in different paths of T and the total memory required by them is smaller than the available memory. An edge (x, y) of the graph expresses that the pipelines in x must be executed before the pipelines in y .

Step 5. For each node of the execution strategy graph, allocate memory for the pipelines and the operations in each pipeline by a heuristic algorithm. The allocation of memory for a pipeline is based on the *data volume ratio of the pipeline* that is the ratio of the data volume processed by the pipeline to the total data volume processed by all the pipeline in the node. The allocation of memory for an operation is based on the *data volume ratio of the operation* that is the ratio of the data volume processed by the operation to the total data volume processed by the pipeline in which the operation is. The allocation of memory is in two levels. At the first level, the memory is allocated to pipelines using the heuristic rule that the whole available memory is shared pro rata among the pipelines in each node of the execution strategy graph according to the data volume ratios of the

pipelines. At the second level, the memory is allocated to the operations in each pipeline using the heuristic rule that the memory allocated for the pipeline at the first level is shared pro rata among the operations in the pipeline according to the volume ratios of the operators.

Step 6. Allocate PC nodes for pipelines and operations. The method is similar to the allocation of memory. The only difference is that the allocation is based on the ratios of the work loads rather than the ratios of the data volumes.

5. APPLICATIONS

Since it was released in 2001, InfiniteDB has been used by many government organizations and enterprises in China, such as the local taxation bureaus in China, China Net Corp., Industrial and Commercial Bank of China, and so on. As an example, we illustrate the Information Management System for Local Taxation Bureaus (IMSforTAX), developed based on InfiniteDB, in sections 5.1, 5.2, 5.3 and 5.4. IMSforTAX has been used by the local taxation bureaus of Heilongjiang province, Guizhou province, and 23 cities in China. In section 5.5, we discuss the challenging problems faced by PC-cluster DBMSs.

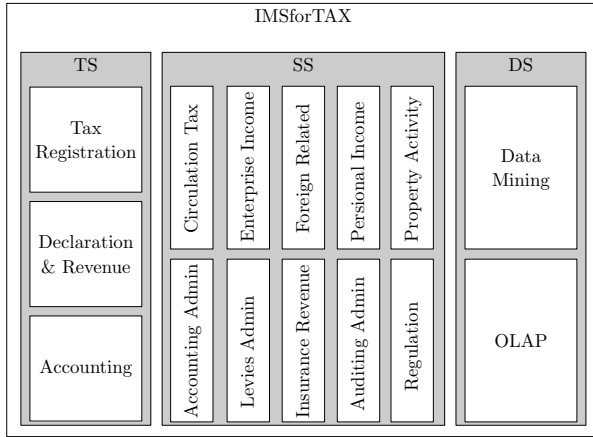


Figure 9: The Structure of IMSforTAX

5.1 Overview of IMSforTAX

As shown in figure 9, IMSforTAX comprises three subsystems, namely the taxation subsystem (TS), the supervision subsystem (SS) and the decision support subsystem (DS).

The taxation subsystem consists of components for handling and inspecting tax registration, declaration & revenue as well as for affiliated accounting issues. The tax registration component processes tax registration, obligor registration, substituted levy registration, etc., and collects related information. By the collected information, the component maintains status about the registration, opening, alternation and cancellation of taxpayers' businesses or enterprises. The tax declaration & revenue component gathers information on taxpayers' financial activities including payment, remittance, underpayment through records and documentations such as tax declaration, payment certification and so forth. The tax accounting component audits the full procedure of tax payment, remittance, underpayment, exemption, and generates certificates, reports, accounting records and statistics of various categories.

The supervision subsystem consists of components for performing supervision concerning circulation tax, enterprise income tax, foreign-related taxation, personal income tax, property activity tax, and tax regulations. The circulation tax component collects and queries basic information about business taxpayers as well as the status of their tax declaration, collection and receiving. The enterprise income tax component collects and queries basic information about enterprise taxpayers as well as their status of the declaration, collection and receiving of corresponding enterprise income tax. The foreign-related taxation component performs information summarization, supervision on foreigners' personal income tax and the audit of foreign-related tax. The personal income tax component collects and queries basic information about personal income taxpayers and the status of their tax declaration, collection and receiving. The property activity tax component inspects the declaration, collection and receiving of property activity tax levied by subordinate tax revenues and performs comparison and analysis. The tax revenue component collects and queries the information collected from the procedures of tax registration, tax declaration & revenue, invoice management and tax auditing. The tax plan & analysis component discovers and generates systematic analysis results reflecting a full perspective on taxation status of present and predictable future based on the raw data collected by subordinate tax administrations and reports derived from these data. The tax auditing component monitors the taxpayers' behaviors, detects their violations of corresponding laws and regulations. The tax regulation component maintains and publicizes the tax regulations announced by provincial taxation bureaus. It supports searching on tax-related regulations in terms of strict keywords matching, hierarchical exploration, combinatorial search, as well as fuzzy search. The insurance revenue component supervises insurance revenue issues.

The decision support subsystem consists of two components specialized for online analysis processing (OLAP) and data mining respectively. The OLAP component organizes taxation data across multiple dimensions based on pre-defined subjects of interest. By doing so, it enables administrators to perform extensive analysis from various perspectives. The data mining component discovers hidden rules of the taxation economy from the collected data, and aids the decision-making of the administrators. Data mining is beneficial in terms of taxation prediction, tax budget making as well as discovering abnormal cases for tax auditing.

To support IMSforTAX, a very large relational database was built. There are five categories of tables in the database, specialized for tax registration, tax declaration & revenue, tax accounting, taxation bureau internals and standard interchange code. The tax registration tables store detailed information on the opening, close out, resume, cancellation and alternation of taxpayers' businesses or enterprises. The tax declaration & revenue tables store the status of tax payment, remittance and revenue. The tax accounting tables store the summarized information for producing accounting reports. The taxation bureau internal tables store the internal affairs of local taxation bureaus including human resource, fixed assets, etc. The standard interchange code tables contain standard codes for information interchange issued by the State Tax Administration, which are recognized among local taxation bureaus and affiliated software packages.

5.2 Requirements of IMSforTAX

The requirements of IMSforTAX depend on the application environments. Here, we take the Local Taxation Bureau of Heilongjiang Province as an example to demonstrate the requirements of IMSforTAX on DBMSs.

The first requirement is the need of massive storage space. There are approximately 17,450,000 taxpayers in Heilongjiang province. 450,000 taxpayers are for enterprise income tax and 17,000,000 taxpayers are for individual income tax. The large number of taxpayers makes the database of IMSforTAX very large. There are more than 600 tables in the database. The total data volume of the tables is more than 1TB per year. The IMSforTAX system is scheduled to operate on 3 years' data. Thus, the required database storage space is more than 3TB.

The second requirement is the need of high transaction processing rate. The expectation number of transactions is approximately 1,300,000 on the busiest day during the revenue period. The 1,300,000 transactions will be processed by 13 IMSforTAXs in 13 cities. Thus, the required transaction processing rate is 16666.6 transactions per hour assuming that there are 6 pure work hours each day.

The third requirement is the need of high reliability. The IMSforTAX system runs 24 hours each day and 7 days each week. It can not suffer data loss during power failure or other hardware malfunctions.

The growth of the tax industry and its emerging business requirements should be taken into consideration while conducting system design. Based on previous statistics, the number of taxpayers grows at a constant rate of 10% each year. Since the IMSforTAX system is started from 2001, the amount of data will reach 2.36T per year and the transaction processing rate will be 39299 transactions per hour by year 2011.

5.3 The Solutions of InfiniteDB for the Requirements of IMSforTAX

We still take the Local Taxation Bureau of Heilongjiang Province as an example to show how InfiniteDB fulfils the requirements of IMSforTAX.

To fulfil the storage space and transaction processing requirements of IMSforTAX, a version of InfiniteDB that runs on a PC-cluster with 32 PC-nodes was installed in the information center of the Local Taxation Bureau of Heilongjiang Province, and 13 InfiniteDBs that run on PC-clusters each with 16 PC-nodes were installed in 13 cities of Heilongjiang province. The IMSforTAXs has been executed on top of the 14 InfiniteDBs. To efficiently process queries on the very large database of IMSforTAX, one of the data declustering methods, CMD, hash, range, round-robin and cloning, was selected for each table in the database according to the characteristics of the queries on the table to minimize the processing time of the queries. For example, the interchange code table contains the collection of standard codes for information interchange. The table is relatively small in size and is rarely updated, but is frequently joined with other tables. To maximize the efficiency of query processing, The cloning method was used to replicate the table among multiple nodes.

To fulfil the reliability requirement, the two levels of fault tolerant and resilient infrastructures of InfiniteDB were employed. All the datasets of the IMSforTAX were protected by the intra-PC node fault tolerant and resilient infrastruc-

ture to guarantee that all the datasets can not suffer data loss during disk failure except the case of all the disks connected to a PC-node being damaged. The inter-PC node fault tolerant and resilient infrastructure was used for protecting important datasets of IMSforTAX to ensure that the important datasets can not suffer data loss during PC-node failure.

The functions of the data warehousing and data mining of the InfiniteDB were employed directly by the IMSforTAX to implement its decision support subsystem.

5.4 Summary

IMSforTAX was designed in full compliance with the requirements of the clients under the support of InfiniteDB. Initiated in early 2001, the IMSforTAX has, until now, been running for 6 years. During this period of time, it delivers impressive performance with high reliability. The short response time for queries and informative results of online analysis processing and data mining significantly enhanced the productivity of affiliated tax administrations. IMSforTAX has received affirmative acknowledgements from a diverse community of taxation officials, tax collectors and taxpayers.

5.5 Challenging Problems

It has been shown in practice that the database size is rapidly growing in many application areas. We tried to catch up with the growing rate of database size by adding PC-nodes to PC-clusters. However, when the number of PC-nodes in a PC cluster is greater than a threshold value, the performance of InfiniteDB goes down due to the communication bottleneck of PC-clusters. Fortunately, we learned that data compression is a feasible technique to improve the performance of InfiniteDB from experiments. The challenging problems are as follows.

- (1). How to develop the data compression methods that are suitable for the random access mode of DBMSs,
- (2). How to develop the algorithms for relational operations and other database operations on compressed relations without the need to decompress the operand relations, and
- (3). How to develop the optimization and processing methods for the queries on compressed relations.

We are currently working on the problems and have proposed data compression methods, which are suitable for the random access mode of DBMSs, and database operation algorithms directly on compressed datasets without the need to decompress them [15, 12].

6. CONCLUSION

We have presented InfiniteDB, a customized PC-cluster DBMS, which is targeted towards parallel processing of massive databases and integrates the functions of data warehousing, data mining and integrating of heterogeneous information resources into its kernel. We have described its data declustering subsystem, its parallel algorithm package for the execution of the relational operations and other database operations, and its query optimization and processing subsystem. In particular we have also discussed the applications of InfiniteDB, especially the IMSforTAX built on InfiniteDB. The applications show that InfiniteDB is an effective and efficient PC-cluster DBMS for storing and processing of massive databases.

From the applications, we learned that data compression

is a feasible technique to improve the performance of PC-cluster DBMSs and it is important to add security mechanism into DBMSs. Now, we are working on data compression methods, database operation algorithms directly on compressed databases without the need to decompression, and the query processing techniques on compressed databases for InfiniteDB. We are also working on adding security mechanism into InfiniteDB.

7. REFERENCES

- [1] Z. Afrookhteh. Technical comparison of oracle real application clusters 10g vs. ibm db2 udb v8.2. [http://www.oracle.com/technology/products/database/clustering/pdf/twp_rac_10g_vs_db2_v8.2\[1\].pdf](http://www.oracle.com/technology/products/database/clustering/pdf/twp_rac_10g_vs_db2_v8.2[1].pdf), August 2005.
- [2] C. K. Baru, G. Fecteau, A. Goyal, H.-I. Hsiao, A. Jhingran, S. Padmanabhan, G. P. Copeland, and W. G. Wilson. Db2 parallel edition. *IBM Systems Journal*, 34(2):292–322, 1995.
- [3] H. Boral, W. Alexander, L. Clay, G. P. Copeland, S. Danforth, M. J. Franklin, B. E. Hart, M. G. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, 1990.
- [4] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, 1990.
- [5] G. Hallmark. Oracle parallel warehouse server. In W. A. Gray and P.-Å. Larson, editors, *ICDE*, pages 314–320. IEEE Computer Society, 1997.
- [6] IBM. Database partitioning feature (dpf). <http://www.tendigit.com/izone/briefs/dpf.html>.
- [7] T. Chen, J. Li. Parallelization techniques for query processing and data declustering methods. *Chinese Journal of Advanced Software Research*, 3(2), 1996.
- [8] J. Li, Z. Cai, and S. Chen. Multi-weighted tree based query optimization method for parallel relational database systems. In H. Lu and S. Spaccapietra, editors, *CODAS*, pages 205–212. IEEE Computer Society, 2001.
- [9] J. Li and W. Du. Parallel cmd-join algorithms on parallel databases. *Chinese Journal of Software*, 9(4):256–262, 1998.
- [10] J. Li, W. jun Sun, and Y. Li. Parallel join algorithms based on parallel b+-trees. In H. Lu and S. Spaccapietra, editors, *CODAS*, pages 197–204. IEEE Computer Society, 2001.
- [11] J. Li and J. Li. A parallel query plan model for parallel relational database systems. *Chinese Journal of Advanced Software Research*, 1(4):301–318, 1994.
- [12] J. Li and J. Srivastava. Efficient aggregation algorithms for compressed data warehouses. *IEEE Transactions on Knowledge and Data Engineering*, 14(3):515–529, 2002.
- [13] J. Li, J. Srivastava, and D. Rotem. Cmd: A multidimensional declustering method for parallel data systems. In L.-Y. Yuan, editor, *VLDB*, pages 3–14. Morgan Kaufmann, 1992.
- [14] M. Stonebraker, R. H. Katz, D. A. Patterson, and J. K. Ousterhout. The design of xprs. In F. Bancilhon and D. J. DeWitt, editors, *VLDB*, pages 318–330. Morgan Kaufmann, 1988.
- [15] W. Wu, H. Gao, and J. Li. New algorithm for computing cube on very large compressed data sets. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1667–1680, 2006.