Ten Thousand SQLs: Parallel Keyword Queries Computing

Lu Qin
The Chinese University of Hong Kong
lqin@se.cuhk.edu.hk

Jeffrey Xu Yu
The Chinese University of Hong Kong
yu@se.cuhk.edu.hk

Lijun Chang
The Chinese University of Hong Kong
ljchang@se.cuhk.edu.hk

ABSTRACT

Keyword search in relational databases has been extensively studied. Given a relational database, a keyword query finds a set of interconnected tuple structures connected by foreign key references. On RDBMS, a keyword query is processed in two steps, namely, candidate networks (*CN*s) generation and *CN*s evaluation, where a *CN* is an SQL. In common, a keyword query needs to be processed using over 10,000 SQLS. There are several approaches to process a keyword query on RDBMS, but there is a limit to achieve high performance on a uniprocessor architecture. In this paper, we study parallel computing keyword queries on a multicore architecture. We give three observations on keyword query computing, namely, a large number of SQLS that needs to be processed, high sharing possibility among SQLS, and large intermediate results with small number of final results. All make it challenging for parallel keyword queries computing. We investigate three approaches. We first study the query level parallelism, where each SQL is processed by one core. We distribute the SQLS into different cores based on three objectives, regarding minimizing workload skew, minimizing inter-core sharing and maximizing intra-core sharing respectively. Such an approach has the potential risk of load unbalancing through accumulating errors of cost estimation. We then study the operation level parallelism, where each operation of an SQL is processed by one core. All operations are processed in stages, where in each stage the costs of operations are re-estimated to reduce the accumulated error. Such operation level parallelism still has drawbacks of workload skew when large operations are involved and a large number of cores are used. Finally, we propose a new algorithm that partitions relations adaptively in order to minimize the extra cost of partitioning and at the same time reduce workload skew. We conducted extensive performance studies using two large real datasets, *DBLP* and *IMDB*, and we report the efficiency of our approaches in this paper.

1. INTRODUCTION

Keyword search on relational databases (*RDB*s) has been extensively studied in recent years. A keyword query is a set of keywords that allows users to find interconnected tuple structures containing the given keywords in an *RDB* [1, 14, 12, 21, 22, 25, 3, 15, 17, 7, 9, 11, 6, 18, 26]. On RDBMS, a keyword query is processed in two steps, namely, candidate networks (*CN*s) generation and *CN*s evaluation. In the *CN*s generation step, it generates all needed *CN*s. In the *CN*s evaluation step, it evaluates all *CN*s using SQLS. The main cost is on the *CN*s evaluation. The challenging is due to a large number of *CN*s to be evaluated. It is common that over ten thousand *CN*s are generated for a keyword query. As an indicator, the number of *CN*s generated for a keyword query can achieve 194,034, as shown in our performance studies. Given such a large number of *CN*s, evaluating each *CN* individually using an SQL is impractical when the size of the *RDB* is large. However, we observe that the *CN*s generated for a keyword query are usually tightly coupled with each other. For example, in the *DBLP* dataset (Fig. 1) with 4 relations, a keyword query of 4 keywords with max 11 tuples in an interconnected tuple structure can generate *CN*s with 539,052 join operations without sharing, and 53,008 join operations after sharing. The probability for any two *CN*s to share computational cost is about 59.64%. In the literature, several algorithms are studied that focus themselves on finding how the computational cost among different *CN*s can be largely shared [14, 22, 27, 25]. However, the problem is still not well solved, because for keyword queries with large number of keywords or with high selective keywords, the processing time for the state of art algorithm is still slow.

In this paper, we study parallel keyword queries computing on a multicore architecture. The main issue discussed is on how to compute an extremely large number of highly coupled SQLS in parallel. In the literature, parallel multiquery processing has been studied in RDBMS. [33] gave an survey of such problems. The existing techniques can not be used to solve our problem for the following two reasons. (1) They focus on allocating a large number of processors to a relatively small number of SQLS, thus each processor only computes a part of an SQL. (2) SQLS do not share computational cost with each other. In our problem, each core needs to process a large number of *CN*s and the portion of the shared computational cost among *CN*s is very large. On a multicore architecture, since the communication cost among cores is small, the main issues making the problem challenging come from three folds. (a) There are always a billion ways to distribute the *CN*s to be processed in different cores. We have to find a way such that *CN*s in the same core share most computational cost and *CN*s in different cores share least computational cost, and at the same time, the workload of different cores are well balanced. (b) We must find an algorithm that allows each *CN* to be processed by multiple cores, with adaptive scheduling in order to handle high workload skew when processing *CN*s. (c) Since the scheduling is needed based on cost estimation, the algorithm must be able to handle the errors caused by estimation adaptively, and avoid errors to be accumulated.

Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

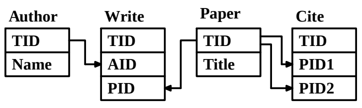


Figure 1: *DBLP* **Database Schema**

The main contributions of this work are summarized below. First, we analyze the parallelism based on three properties of the keyword query computing, namely, a large number of SQLs, large sharing among SQLs, and a large number of intermediate tuples generated. Second, we propose three approaches to compute keyword queries in parallel on a multicore architecture. The first approach distributes *CN*s into different cores that achieves three objectives, namely, minimizing workload skew, minimizing inter-core sharing, and maximizing intra-core sharing. We then show that the first approach is error sensitive, and propose the second approach that allows each *CN* to be processed in different cores adaptively. We allow staging in this approach, where in each stage all cores only read the shared memory allocated in previous stages and write results to their allocated new memory individually. The memory in the previous stages will be freed as soon as it will not be used anymore. We show that the second approach can largely reduce the accumulated error caused by the first approach as well as reduce the memory used by the large intermediate result. We also show that the second approach can not handle well when the workloads of operations in *CN*s are highly skewed. We propose the third approach to allow each relation to be adaptively partitioned and processed in different cores. Using the third approach, we can limit the number of partitions used, reduce the sensitivity of estimation error, and achieve a good approximate bound regarding workload balancing. Finally, we conducted extensive performance studies using two large real datasets and confirmed the efficiency of our approaches. We can improve the performance of the state of art sequential algorithm to be 10 times faster on average using 16 cores.

The remainder of the paper is organized as follows. In Section 2, we give the formal problem definition. In Section 3, we show the state of art sequential algorithm to solve the problem. In Section 4, we analyze the properties of the problem to be parallelized. In Section 5, Section 6, and Section 7, we discuss three approached with different levels of parallelism. We discuss the related work in Section 8. We conducted experimental studies and discuss our findings in Section 9, and conclude our paper in Section 10.

2. PROBLEM DEFINITION

We consider a database schema in a relational database as a directed graph $G_S(V, E)$, called a schema graph, where V represents the set of relation schemas $\{R_1, R_2, \cdots\}$ and E represents foreign key references between two relation schemas. Given two relation schemas, R_i and R_j, there exists an edge in the schema graph, from R_i to R_j, denoted $R_i \rightarrow R_j$, if the primary key defined on R_i is referenced by the foreign key defined on R_j. Parallel edges may exist in G_S if there are several foreign keys defined on R_j referencing to the primary key defined on R_i. To distinguish one foreign key reference among many, we use $R_i \xrightarrow{X} R_j$, where X is the foreign key attribute name. In a relation schema R_i, we call an attribute, defined on strings or full-text, a text attribute, to which keyword search is allowed. A relation on relation schema R_i is an instance of the relation schema (a set of tuples) conforming to the relation schema, denoted $r(R_i)$. Below, we use R_i to denote $r(R_i)$ if the context is obvious, and we use $V(G_S)$ and $E(G_S)$ to denote the set of nodes and the set of edges of G_S, respectively. A relational database (*RDB*) is a collection of relations.

A simple *DBLP* database schema, G_S, is shown in Fig. 1. It consists of four relation schemas: Author, Write, Paper, and Cite.

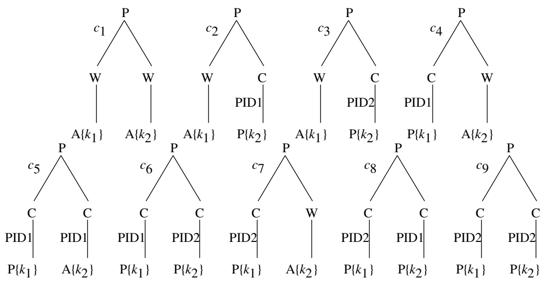


Figure 2: Nine ***CN*****s**

Each relation has a primary key TID. Author has a text attribute Name. Paper has a text attribute Title. Write has two foreign key references: AID (refer to the primary key defined on Author) and PID (refer to the primary key defined on Paper). Cite specifies a citation relationship between two papers using two foreign key references, namely, PID1 and PID2 (paper PID2 is cited by paper PID1), and both refer to the primary key defined on Paper.

An m-keyword query is a set of keywords of size m, $\{k_1, k_2, \cdots, k_m\}$. An answer of an m-keyword query is a minimal total joining network of tuples, denoted *MTJNT* [14, 22, 25, 27, 32]. First, a joining network of tuples (*JNT*) is a connected tree of tuples where every two adjacent tuples, $t_i \in r(R)$ and $t_j \in r(R')$ can be joined based on the foreign key reference defined on relational schemas R_i and R_j in G_S (either $R_i \rightarrow R_j$ or $R_j \rightarrow R_i$). Second, by total, it means that an *MTJNT* must contain all the m keywords. Third, by minimal, it means that an *MTJNT* is not total if any tuple is removed. The minimal condition implies that every leaf tuple in the tree must contain at least one keyword. The size of an *MTJNT* is the number of tuples in the tree, and a user-given parameter Tmax is used to specify the maximum number of tuples in *MTJNT*s, in order to avoid an *MTJNT* to be too large in size, because it is not meaningful if two tuples are connected by a long chain of tuples.

The problem of m-keyword query processing we study in this paper is to find all *MTJNT*s of size $\leq \mathsf{Tmax}$, for a given m-keyword query, $K = \{k_1, k_2, \cdots, k_m\}$, over a schema graph G_S on a multicore system. In the framework of RDBMS, the two main steps of processing an m-keyword query over a schema graph G_S are candidate network generation and candidate network evaluation. In the first candidate network generation step, it generates a set of candidate networks over G_S, denoted $C = \{c_1, c_2, \cdots\}$, to be evaluated in the second step. In brief, a candidate network (*CN*), c_i, corresponds to a relational algebra that joins a sequence of relations with selections of tuples for keywords over the relations involved. The set of *CN*s shall be sound/complete and duplicate-free. The former ensures all *MTJNT*s must be found, and the latter is mainly for efficiency consideration. In the second candidate network evaluation step, all $c_i \in C$ generated will be evaluated. Fig. 2 shows nine *CN*s for the keyword query $K = \{k_1, k_2\}$ in *DBLP* with $\mathsf{Tmax} = 5$. For simplicity, we use A, W, P and C to denote the relations Author, Write, Paper and Cite respectively, and use $R\{K'\}$ to denote the subrelation $\{t | t \in R \wedge \forall k \in K', t \text{ contains } k \wedge \forall k \in (K - K'), t \text{ does not contain } k\}$. In Fig. 2, the first *CN* c_1 denotes the join sequence $A\{k_1\} \bowtie W \bowtie P \bowtie W \bowtie A\{k_2\}$.

3. STATE OF ART

Regarding *CN* generation, in the literature, there are many fast algorithms proposed to reduce the computational cost by eliminating the number of intermediate partial *CN*s generated [14, 22, 27]. Since the set of *CN*s are only related to the schema graph G_S and the number of keywords m in the query, all *CN*s can be computed offline. In this paper, we focus on improving the performance of the second online *CN* evaluation step using multicore systems.

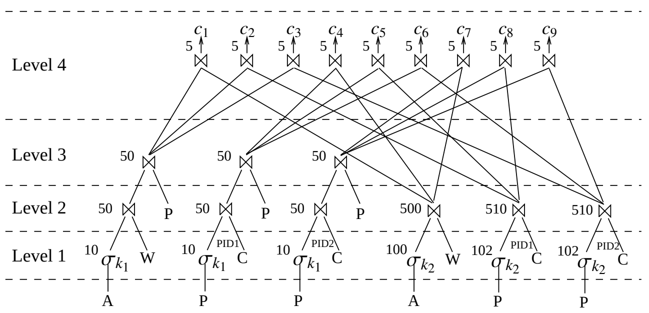


Figure 3: A Partial Execution Graph for Query $\{k_1, k_2\}$

We first give a brief introduction to the state-of-art sequential algorithm to evaluate all *CN*s, which is also the baseline algorithm to be compared. The problem for *CN* evaluation is a multi-query optimization problem in the framework of RDBMS, where a large number of *CN*s should be evaluated all together in order to minimize the overall computational cost. As discussed in DISCOVER [14], it is an NP-complete problem to generate the optimal plan to evaluate all *CN*s. It is different from the single query optimization problem because a subexpression in a *CN* can be largely shared by other *CN*s. Optimal plans for each individual *CN* can not always yield global optimal. Fig. 13 in Appendix A shows an example to explain this.

Generally speaking, before evaluating a subexpression E, two values should be considered, namely, the estimated number of tuples of E, denoted *size*, and the number of *CN*s that share E, denoted $freq$. A score function $score(E) = f(size, freq)$ is defined on E which increases with $freq$ and decreases with *size*. When evaluating all *CN*s, the subexpression that yields the largest score should be evaluated iteratively until all *CN*s has been evaluated. As discussed in [14], the score function $score(E) = \frac{freq^a}{log^b(size)}$ can yield a near-optimal execution plan where a and b are constants.

Execution Graph: Similar to the single query optimization problem, the join plan of the multi-query optimization problem can be represented by a graph called the execution graph, denoted $G_E(V, E)$. In G_E, each node $v \in V(G_E)$ is an operator (e.g. $\bowtie$ or σ), representing the root of the corresponding join subsequence in the execution plan. There is an edge from node v_1 to node v_2 in G_E (i.e., $(v_1,v_2) \in E(G_E)$) if and only if the output of v_2 is part of the input of v_1 in the execution plan. We call v_1 a father node of v_2 and v_2 a child node of v_1. G_E is a directed acyclic graph (DAG) where each node may have multiple father nodes and multiple child nodes. For each node $v \in V(G_E)$, we denote the set of father nodes of v by $fnodes(v)$ and denote the set of child nodes of v by $cnodes(v)$. Note that the leaf nodes of the execution plan are relations in the original database other than operators, thus they are not part of $V(G_E)$. There are several levels in G_E, which are defined recursively as follows. (1) All nodes that represent the selection of keywords are in level 1. (2) For each level l ($l > 1$), a node v is in level l iff $\exists\, v'$, such that $(v,v') \in E(G_E)$ and v' is in level $l - 1$, and $\forall v''$, such that $(v,v'') \in E(G_E)$, the level of v'' is no larger than $l - 1$. The maximum level of G_E is the depth of G_E and is denoted as $depth(G_E)$. For each node $v \in V(G_E)$, there is a estimated cost, denoted $cost(v)$, and the total cost of evaluating all *CN*s is denoted as $cost(G_E)$ which can be calculated as $\sum_{v \in V(G_E)} cost(v)$. In the state of art algorithms [14, 22, 25, 27], each node in $V(G_E)$ is evaluated in a bottom-up fashion sequentially, and there is no parallelism involved.

Example 3.1: Fig. 3 shows the execution graph G_E for the nine queries listed in Fig. 2. It has four levels. Each node in level 3 has 3 father nodes and each node in level 4 has 2 child nodes. For the leftmost node in level 4, it represents the output of the *CN* $c_1 = (A\{k_1\} \bowtie W \bowtie P) \bowtie (W \bowtie A\{k_2\})$. It has two child nodes, representing the output of the subsequences $A\{k_1\} \bowtie W \bowtie P$ and $W \bowtie A\{k_2\}$ respectively. We also mark the estimated cost beside each node in Fig. 3. The total cost for evaluating the nine *CN*s is $cost(G_E) = \sum_{v \in V(G_E)} cost(v) = 2199$. □

4. ANALYSIS FOR PARALLELISM

Parallel query processing in relational databases has been extensively studied for many years. [20] introduces many algorithms for parallel query processing in RDBMS. In the literature, the methods used on query parallelism can be divided into three categories, namely, intra-operator, inter-operator, and inter-query parallelism. A survey for the three kinds of parallelism is given in [33]. Intra-operator parallelism performs a certain relation operation in parallel. It focuses on splitting an operation into subtasks in a manner that the workload is balanced among a given number of processors. Inter-operator parallelism performs an SQL query in parallel. It can be achieved in two steps. The first step generates the execute plan tree, such that operations that can be performed in parallel are identified. The second step allocates a certain number of processors to each operation for workload balancing. Inter-query parallelism focuses on processing multiple queries simultaneously. The main issue is also processor allocation such that costly queries are allocated with more processors to reduce the workload skew.

In this paper, we focus on parallel keyword search, a certain type of multiquery optimization problem. Our problem have the following properties.

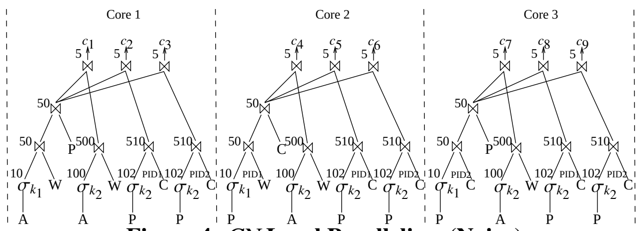
(1) Large number of queries: In the traditional inter-query parallelism methods, it assumes that the number of SQL queries executed is not large such that each query can be allocated a certain number of processors. In our problem, the number of SQL queries to be evaluated simultaneously is extremely large. For example, in our experiments, the number of SQL queries/*CN*s generated by a certain keyword query can achieve 194,034. In such cases, even a scheduling method with time complexity $O(m^2)$ is unacceptable where m is the number of queries generated.

(2) Large sharing between queries: The traditional inter-query parallelism algorithms do not consider sharing among multiple SQL queries to be evaluated simultaneously, since queries are always independent with each other. For our problem, instead, we speed up the state of art sequential algorithm by taking sharing into consideration. Such sharing part is extremely large because all SQL queries are tightly coupled with each other. Take the simple *DBLP* database (Fig. 1) for example, for a keyword query with 4 keywords and the size control parameter Tmax be 11, there will be $539,052$ join operations in total when sharing among *CN*s is not considered. Such number can be reduced to $53,008$ when subexpression sharing is taken into consideration. Given any two *CN*s randomly, with probability 59.64%, they will share at least one common subexpression. The expected sharing between any two randomly selected *CN*s is 1.235 join operations. Algorithms without considering sharing among *CN*s need to process 10 times more operations.

(3) Large intermediate result: An important issue for our problem is that the intermediate result generated by the internal operators is extremely large, and the final result size is relatively much smaller. For example, in the *DBLP* database with schema shown in Fig. 1, for a keyword query $\{keyword, relation, parallelism\}$ and size control parameter Tmax be 9, using the state of art sequential algorithm, it will generate $379,017$ intermediate tuples while the final result only contains 272 tuples. For each such internal operator, the intermediate result is considered as input of many other operators. In our experiments, an internal operator can be shared by more than 1,000 other operators as input. Such large cross-sharing among operators make it difficult to make good use of the intra-operator parallelism to divide relations and inter-operator parallelism for scheduling and pipelining using the left/right deep trees or bushy trees.

We now show that the traditional parallel query processing techniques on all three kinds of parallelism can not be easily extended to solve our problem. (1) The inter-query parallelism is most relevant to our problem to process multiple SQLs. In the traditional inter-query parallelism problem, it implicitly assumes that the number of queries is no larger than the number of processors used, while in our problem we assume that the number of *CN*s generated is much larger than the number of cores used for processing. In such a situation, the main concern is not just allocate cores to *CN*s, but allocate *CN*s to cores such that each core can process multiple *CN*s for workload balancing. Further more, the traditional inter-query parallelism problem does not consider sharing among SQLs, while in our problem, subexpressions in *CN*s are largely shared by each other. It makes the problem more difficult when allocating *CN*s to differen cores. (2) The inter-operator parallelism processes a certain query using multiple processors. We can also consider all *CN*s as a large SQL query, and process the large SQL query using the inter-operator parallelism techniques. It is difficult because in the inter-operator parallelism the execute plan of an SQL is a tree, and all operations can be processed in parallel because sibling operations can be processed individually and parent-child operations can be paralleled through pipelining. In our problem, the large sharing among *CN*s makes the overall execute plan a graph instead of a tree, and the number of operations to be performed is much larger than the number of cores used. How to allocate operations to different cores is not considered in inter-operator parallelism. (3) The intra-operator parallelism allows each operator to be processed using different processors. For our problem, we can change the sequential algorithm to perform each operator using the intra-operator parallelism techniques. In this way, dividing and merging the intermediate relations becomes the bottleneck of the problem because the number of intermediate tuples is very large in our problem.

In order to solve the above problems, instead of allocating cores to *CN*s, we give several new approaches. Our first attempt is to partition *CN*s to be processed in different cores based on the cost estimation function of each operation in the *CN*. Our aim is to balance the workload of each core, and under that condition, maximize the intra-partition sharing of *CN*s in each core and minimize the inter-partition sharing of *CN*s in different cores. We call this approach the *CN* level parallelism and will discuss it in Section 5. This approach is sensitive to the accuracy of the cost estimation function used because of the error accumulation when estimating a sequence of join operations all together. Our second attempt improves the previous algorithm in two ways. First, we allow each *CN* to be processed in multiple cores, such that each *CN* can share part of the result generated by other *CN*s in order to reduce the overall computational cost. Second, we allow changing the scheduling adaptively by re-estimating the computational cost of each operation to be processed. We only allow one-level estimation cost to be used, which can largely reduce the accumulated error caused by sharing. In the cost estimation function, all cost values involved in calculation must be the real cost other than the estimated cost. In our algorithm, we use several stages, and in each stage, all cores only read the shared memory allocated in previous stages and write results to their allocated new memory individually. The memory in previous stages will be freed as soon as it will not be used anymore, in order to reduce the memory used by the large intermediate result. We call such an approach the operation level parallelism and will discuss it in Section 6. Finally, we found that in some situations, few operations can be very costly such that even the operation level parallelism can not find good balancing among cores. We further propose the data level parallelism that allows an operation to be processed by multiple cores through partitioning of relations. We show that using our approach, the relations can be partitioned adaptively to minimize the workload skew when processing, and further more, the sensitivity of estimation error can be further reduced through data level partitioning. We will discuss data level parallelism in Section 7.



Core 1
Core 2
Core 3
c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9
5 50 500 510 10 100 102 PID
σ_{k_1} σ_{k_2}
A W P C

Figure 4: *CN* Level Parallelism (Naive)

5. *CN* LEVEL PARALLELISM

In this section, we assume that each *CN* must be processed in one core, and thus divide *CN*s into different partitions such that *CN*s in the same partition are processed by a single core. In such a case, subexpressions in a certain core can be shared using existing algorithms, but subexpressions in different cores can not be shared by each other. As a result, some redundant work will be performed in different cores. We first give a straightforward solution followed by the discussion of our new algorithm.

A Straightforward Approach: We consider each *CN* individually first, and then distribute them to partitions such that *CN*s in each partition are processed by a single core. The only objective is to balance the total cost for *CN*s processed in each core. This is a simple approach, and we do not need to consider intra-partition sharing and inter-partition sharing when scheduling. We only need to follow the generally used largest first rule for workload balancing that large tasks should be distributed first and should be assigned to the partition with the least workload. The algorithm for the naive approach is shown in Algorithm 1 in Appendix B.

Theorem 5.1: *The time complexity of Algorithm 1 for scheduling is $O(m \cdot (n + \mathsf{Tmax} + \log m))$.* □

The proof sketch is given in Appendix B.

Example 5.1: Fig. 4 shows the scheduling using 3 cores for the same problem in Example 3.1. The cost for all partitions are the same (well balanced), which is 1949, thus the final cost is 1949. Comparing to the cost 2199 of the sequential algorithm (Example 3.1), it is 88.6% of the sequential cost. □

Sharing-Aware *CN* Partitioning: The performance of Algorithm 1 is not good because it does not consider the subexpression sharing in each core and cross different cores. An ideal algorithm should satisfy the following three objectives: (1) Minimizing workload skew: This is to avoid a large partition dominant other partitions and thus becomes the bottleneck. (2) Maximizing intra-partition sharing: This is used to reduce the processing time of individual cores. Because given the same number of *CN*s in a certain core, the larger they share computational cost, the better the performance is. (3) Minimizing inter-partition sharing: This is used to reduce the number of redundant works processed by different cores.

In our algorithm, we make improvements on the following aspects in order to satisfy the three objectives. (1) The priorities of the

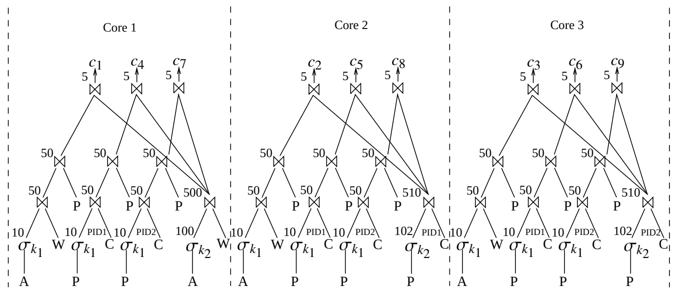


Figure 5: *CN* Level Parallelism (New)

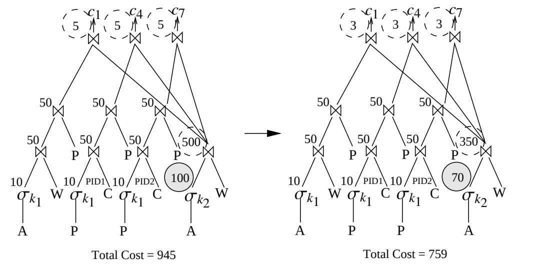


Figure 6: An Error of 30 Induces an Accumulated Error of 186

*CN*s to be distributed are changed over time, according to the cost of the not-shared/extra part of each *CN*. The *CN* with the largest not-shared/extra cost is distributed first in order to minimize the workload skew. (2) The total cost for a partition is recomputed as the cost after sharing subexpressions for all *CN*s in that partition. (3) The *CN* selected for distributing is allocated to the partition with maximum sharing if it does not destroy the workload balancing. Our new algorithm is shown in Algorithm 2 in Appendix B.

Theorem 5.2: *The time complexity of Algorithm 2 for scheduling is* $O(m \cdot \log m \cdot \mathsf{Tmax} \cdot n)$. □

The proof sketch is given in Appendix B.

Example 5.2: For the same problem in Example 3.1, suppose there are 3 cores and suppose it is in the third iteration. The *CN* selected to be distributed is c_7, and the partition selected is partition 1. After removing the execute plan tree of c_7, the costs for some other *CN*s should be updated. Fig. 14 in Appendix B illustrates the process of updating the costs. Fig. 5.2 shows the scheduling of the example using Algorithm 2. The cost for the 3 cores are 945, 957 and 957 respectively and the final cost is 957. Comparing to the cost 2199 of the sequential algorithm (Example 3.1), it is 43.5% of the sequential cost, which is much better than the result shown in Example 5.1. □

6. OPERATION LEVEL PARALLELISM

In the *CN* level parallelism, there are two main drawbacks: (1) redundant works are done by multiple cores because each *CN* must be processed in one core and different *CN*s in different partitions cannot share subexpressions. (2) The static scheduling according to cost estimation of *CN*s is very sensitive to the error caused by the inaccuracy of the cost estimation function. This is because the result of the inaccurate estimation will be used as part of the estimation function in many operations in higher levels. The lower the level of the estimated operation is, the larger accumulated error it causes. As an example, Fig. 6 (left) shows the execute plan of Example 5.2 on the first core using Algorithm 2. Suppose now the cost 100 in the gray circle is reduced to 70 due to estimate inaccuracy. The new cost for each operation is shown in the right hand side. The total cost for all operations is reduced from 945 to 759. As a result, an error of 30 induces 6 times more accumulated error, which makes the total workload unbalanced.

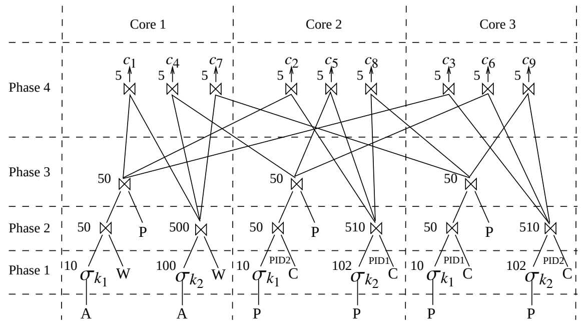


Figure 7: Operation Level Parallelism

In this section, we study the operation level parallelism, where each *CN* is allowed to be processed in different cores, but each operation (e.g. join) in the execute graph of *CN*s must be processed in a certain core. Since a subexpression E evaluated in a core can be reused when evaluating other *CN*s that share E in other cores. The other cores have to wait until the evaluation of subexpression E in its core has been finished. We divide all nodes in the execute graph G_E into several phases/stages, and a node $v \in V(G_E)$ will be processed in phase l ($1 \leq l \leq depth(G_E)$) iff v is in level l of G_E. In this way, nodes in the same phase can be processed independently, and nodes in higher phases need the result generated by the nodes in lower phases. All nodes in the same phase can be scheduled adaptively according to the largest first rule. In each phase, all cores only read the shared memory allocated in previous phases and then write results to their allocated new memory individually. The memory in previous phases will be freed as soon as it will not be used anymore, in order to reduce the memory used by the large number of intermediate tuples. In order to eliminate the error accumulation effect, before each phase, we re-estimate the cost of each operation in the phase using more accurate statistics of results obtained in the previous phases. There are two kinds of operations. The first one is the form $v = R\{K\}$, which is a selection of keywords in K from a certain relation R in the *RDB*. In such a situation, the cost of v can be calculated as follows.

$$cost(v) = |R| \times \Pi_{k \in K} sel(k) \tag{1}$$

where $sel(k)$ is the selectivity of the keyword k in the relation R which can be calculated as $sel(k) = \frac{|R\{k\}|}{|R|}$, and can be pre-computed and saved beforehand. Here $|R\{k\}|$ denotes the number of tuples in R that contain the keyword k. The second kind of operation is the join operation, in the form $v = R.a_i \bowtie R'.a_j$. Suppose the primary key a_i is referenced by the foreign key a_j in the original database. The cost of v can be calculated as follows.

$$cost(v) = |R| \times |R'| \times sel(a_i \rightarrow a_j) \tag{2}$$

where $sel(a_i \rightarrow a_j) = \frac{|rel(a_i) \bowtie rel(a_j)|}{|rel(a_i)| \times |rel(a_j)|}$, and can also be pre-computed and saved beforehand for each edge $(rel(a_i), rel(a_j)) \in E(G_S)$. Here $rel(a_i)$ and $rel(a_j)$ are the relations in the original database where a_i and a_j come from.

The detailed algorithm is shown in Algorithm 3 in Appendix C.

Theorem 6.1: *The time complexity of Algorithm 3 for scheduling is* $O(\mathsf{Tmax} \cdot m \cdot (\log \mathsf{Tmax} + \log m))$. □

The proof sketch is given in Appendix C.

Below, we use Example 6.1 to show that operation level parallelism (Algorithm 3) can reduce the computational cost of all cores, and use Example 6.2 to show that operation level parallelism can reduce the error caused by accumulation.

Example 6.1: For the same problem in Example 3.1, suppose there

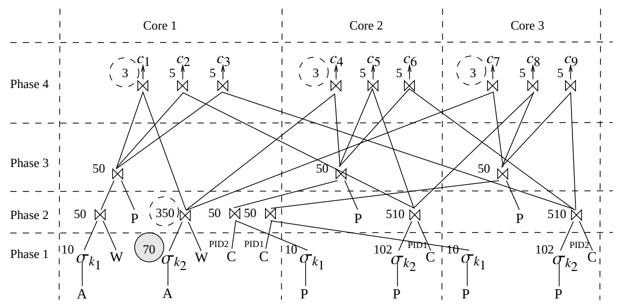


Figure 8: Adaptive Scheduling

are 3 cores, the scheduling using Algorithm 3 is shown in Fig. 7. The costs for the 3 cores are 725, 737 and 737 respectively and final cost is 737. Comparing to 2199 in Example 3.1, it is 33.5% of the sequential cost, which is almost optimal, since the lower bound of the cost using 3 cores is 33.3%. □

Example 6.2: Suppose that the cost 100 of the operation in level 1 is reduced to 70 because of inaccuracy of estimation as denoted in Fig. 6. Using Algorithm 3, the adaptively changed scheduling is shown in Fig. 8. Nodes in phase 2-4 change positions according to the new estimated cost for operations. Using this way, the costs for the 3 cores are 643, 685 and 685 respectively. The workloads are well balanced, and the final cost is 685. Comparing to the result in Fig. 6 and Fig. 7, instead of reducing the cost of only one core, the costs of all cores are reduced using the adaptive scheduling. □

7. DATA LEVEL PARALLELISM

The operation level parallelism performs well when the costs for all join operations in the same phase vary not too much. In practice, it is possible that a certain join operation is much more costly than others, and becomes the dominant cost when processing. In such a case, operation level parallelism does not scale well when the number of cores increases. In this section, we study a smaller granularity of parallelism, namely, the data level parallelism, where each operation in G_E can be performed on multiple cores.

Adaptive partitioning: We propose a new strategy to handle workload skew, that uses the operation level parallelism if there is no workload skew, and partition data adaptively before each time workload skew happens. In such strategy, the following three subproblems have to be solved. (1) Choose the appropriate phase and appropriate nodes for partitioning. (2) Choose the appropriate phase and appropriate nodes for merging. (3) Choose an appropriate partitioning algorithm. For the first subproblem, in each phase, we choose the most costly node to be partitioned, because such node is the root cause of workload skew. For the second subproblem, we choose to merge the result in the final phase. This is because merging results will induce extra cost especially when the number of tuples in each result is large. For the property of our problem, the number of intermediate tuples is much larger than the final output, so merging in the final phase will induce least merging cost. For the third problem, we use the adaptive partitioning strategy that iteratively partition the most cost node in each phase until the workload skew is small. It is worth noting that once a node in a certain phase is partitioned, the execute graph should also be changed, by adding a copy of the partitioned node, as well as copies of all nodes in higher phases that share it. Our algorithm is shown in Algorithm 4 in Appendix D.

Remark 7.1: *Consider the estimation error as a random variable, we model the sensitivity of estimation error as the standard deviation of the estimation error. Adaptive data partitioning decreases the sensitivity of estimation error.* □

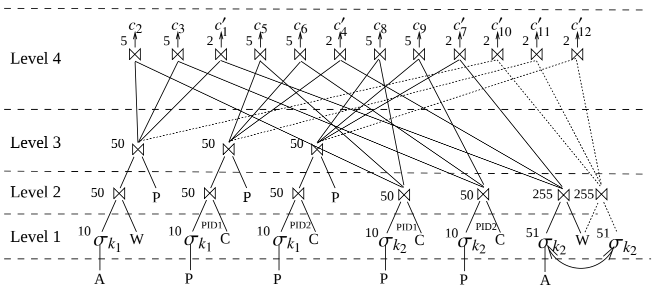


Figure 9: Data Level Parallelism

Remark 7.1 shows that adaptive data partitioning not only decreases the workload skew, but also decreases the sensitivity of estimation error no matter whether workload skew happens or not. We show this using a simple example. We model the estimated cost of every operation as a random variable satisfying normal distribution. Now consider 4 variables X_1, X_2, X_3 and X_4, all satisfying $N(\mu, \sigma^2)$. Here μ can be considered as the estimated cost of every operation and σ^2 can be considered as the sensitivity of the error for estimation. X_1 and X_2 are partitioned from the same operation, and X_1, X_2 are independent of X_3 and X_4, and X_3 and X_4 are independent with each other. Without partitioning, X_1 and X_2 are combined as a single operation, and X_3 and X_4 are put together. The workload skew can be modeled as $Y_1 = (X_1 + X_2) - (X_3 + X_4)$. After partitioning, X_1 and X_2 are put into different groups, and X_3 and X_4 are also put into different groups. The new workload skew can be modeled as $Y_2 = (X_1 + X_3) - (X_2 + X_4)$. Since X_1 and X_2 are partitioned from the same operation, they are highly correlated. We have $Y_1 \sim N(0, 6\sigma^2)$ and $Y_2 \sim N(0, 2\sigma^2)$. Obviously, although the estimated workload skew are all 0, the sensitivity using partitioning is much smaller. For other kinds of distributions, we can also derive similar results.

Lemma 7.1: *In each phase of Algorithm 4, at most $n - 1$ partition operations will be performed.* □

Theorem 7.1: *Suppose the optimal solution is the case where the estimated workloads are evenly divided among all cores. Algorithm 4 is a 2-approximate algorithm.* □

Theorem 7.2: *The time complexity of Algorithm 4 for scheduling is $O(\mathsf{Tmax}^2 \cdot m \cdot n \cdot (\log n + \log \mathsf{Tmax} + \log m))$.* □

The proof sketch of Lemma 7.1, Theorem 7.1 and Theorem 7.2 are all given in Appendix D.

Example 7.1: For the same problem in Example 3.1, suppose there are 3 cores, and the costs for the last two nodes (with cost 102) in level 1 are changed to 10. Suppose we are currently in phase 2, where workload skew happens because the node with cost 502 become the bottleneck. We need to divide one of its child nodes. Fig. 9 illustrates the execute graph after partitioning the node with cost 102 in level 1 into two nodes with cost 51 each. The 3 copied nodes, c'_{10}, c'_{11} and c'_{12} are added in level 4, regarding *CN*s c_1, c_4 and c_7 respectively, because c_1, c_4 and c_7 all share the partitioned node. After the partitioning, the two operations with cost 255 can be distributed into the first two cores and all other 5 operations with cost 50 can be distributed into the third core in level 2. The final cost is $max\{250, 255, 255\} = 255$, which is well balanced. □

8. RELATED WORK

Query Processing in Relational Databases: Query processing and query optimization in relational databases have been extensively studied. Some surveys can be found in [23, 10, 5, 31]. Multi-queries optimization were also studied in [28, 34, 16] by sharing common subexpressions among queries or inside a query. Query processing in parallel relational database systems is most related to our work and has been discussed in the paper. Lu et al. [20] gave a general introduction to such kinds of problems.

Parallel Programming: Multiprocessor scheduling has been studied extensively in parallel programming. In multiprocessor scheduling, the precedence constraints of tasks are also modeled as a DAG [4, 30], and many other such problems are also surveyed in [4]. The multiprocessor scheduling problem is different from the problem studied in this paper mainly because of the following three reasons. (1) In multiprocessor scheduling, the sensitivity of estimation error and accumulated error is not considered. (2) In multiprocessor scheduling, each node in a DAG can not be further divided. In our problem, we allow data level parallelism where each node in the execute graph can be further divided adaptively to reduce the workload skew. (3) In multiprocessor scheduling, the time complexity of the algorithms in the literature vary from $O(\mathcal{N}^2)$ to $O(\mathcal{N}^4)$, where $\mathcal{N}$ is the number of nodes in the DAG. In our problem, the number of nodes in the DAG is extremely large, and even $O(\mathcal{N}^2)$ is not practical for our settings. For multicore programming, algorithms are studied for different applications. [24] and [29] studied skyline computation and tree-structured data mining in a multicore system respectively. The problems studied did not have the three properties of the problem discussed in this paper. Many softwares like Map/Reduce, Hadoop, and Dryad are also developed for parallel computing. They did not consider the schema based adaptive scheduling as well as the three properties for the problem.

Keyword Search in Relational Databases: The related work on keyword search on relational databases include [1, 14, 12, 21, 22, 2, 13, 19, 3, 15, 17, 7, 9, 11, 6, 18, 26, 25]. Discussions can be found in Appendix E.

9. PERFORMANCE STUDIES

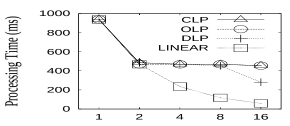
We conducted extensive performance studies to test the algorithms proposed in this paper. We implemented four algorithms, Algorithm 1, Algorithm 2, Algorithm 3, and Algorithm 4, denoted CLP-Naive, CLP, OLP and DLP respectively. For Algorithm 1, since the inter-core sharing among cores is very large, the performance is almost the same to the sequential algorithm when the number of cores increases in all of our testings. We will not include the CLP-Naive algorithm in our experiments. We also add another curve, denoted LINEAR, in our testings, which is processing time using multiple cores with linear speedup. For any test using n cores, the LINEAR processing time is calculated as $t_{linear} = \frac{t_{sequential}}{n}$ where $t_{sequential}$ is the processing time for the state of art sequential algorithm as introduced in Section 3. All algorithms were implemented in Visual C++ 2008 and all tests were conducted on a PC with 4 quad core 2.26GHz Inter(R) Xeon(R) processor and 4GB main memory running Windows Server 2003.

We use two large real datasets, *IMDB* (up to Feb. 2010)[1] and *DBLP* (up to Feb. 2010)[2] for testing. We report the testing results using *IMDB*, and the details for *DBLP* can be found in appendix F.

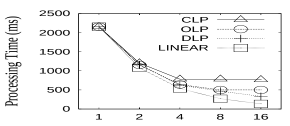
For the *IMDB* dataset, the schema includes the following eight relations: Actor(<u>Actorid</u>, Actorname), Actorplay(<u>Actorplayid</u>, Actorid, Movieid, Charactor), Director(<u>Directorid</u>, Directorname), Direct (<u>Directid</u>, Directorid, Movieid), Movie(<u>Movieid</u>, Name), Genres (<u>Genreid</u>, Movieid, Genre), Actress(<u>Actressid</u>, Acttressname), Actressplay(<u>Actressplayid</u>, Actressid, Movieid, Charactor). The primary key for each relation is underlined. The size of the raw data for the *IMDB* dataset is 956MB. The number of tuples in the eight relations are 1,153,978, 8,192,168, 188,909, 1,065,455, 1,554,266, 986,844, 675,183 and 4,768,278 respectively, and the total number of tuples in *IMDB* is 18,585,081.

[1]http://www.imdb.com/interfaces

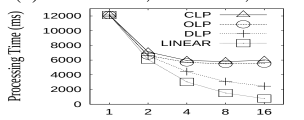
[2]http://www.informatik.uni-trier.de/~ley/db/



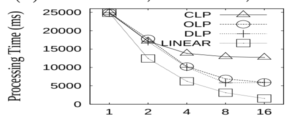
(a) Tmax = 4, #CN = 1,884



(b) Tmax = 5, #CN = 5,093

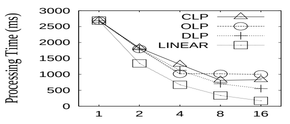


(c) Tmax = 6, #CN = 11,809

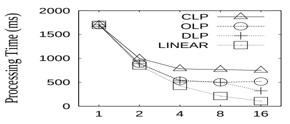


(d) Tmax = 7, #CN = 26,642

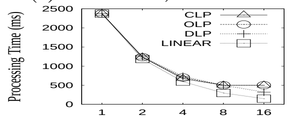
Figure 10: Vary Tmax (*IMDB*)



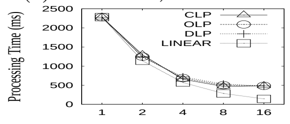
(a) Knum = 2, #CN = 94



(b) Knum = 3, #CN = 735



(c) Knum = 5, #CN = 32,432



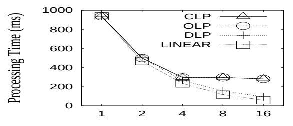
(d) Knum = 6, #CN = 194,034

Figure 11: Vary Knum (*IMDB*)

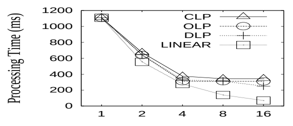
For all testings, we report the processing time for each test case using 1, 2, 4, 8 and 16 cores. The processing time includes the time for scheduling and parallel processing. For each dataset, we select representative queries with different keyword selectivity as follows. After removing all the stop words, we set the maximum keyword selectivity among all keywords as τ, and divide the keyword selectivity range between 0 and τ into 5 partitions, namely, $\tau/5$, $2\tau/5$, $3\tau/5$, $4\tau/5$ and τ. For simplicity, we say a keyword has selectivity p ($p \in \{1, 2, 3, 4, 5\}$), if and only if its selectivity is between $(p-1) \cdot \tau/5$ and $p \cdot \tau/5$.

We vary 3 parameters, namely, the average keyword selectivity Ksel, the keyword number Knum, and the size control parameter Tmax. Every parameter has a default value. For *IMDB*, the keyword selectivity Ksel ranges from 1 to 5 with a default value 3. The keywords selected with different keyword selectivity are shown in Table 1 in Appendix F. The keyword number Knum ranges from 2 to 6 with a default value 4, and the size control parameter Tmax ranges from 4 to 7 with a default value 5. When varying keyword number Knum, we select the first Knum keywords from the keywords with the default selectivity.

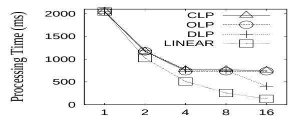
Exp-1 (Vary Tmax): Fig. 10 shows the performance for all algorithms when Tmax changes from 4 to 7 in the *IMDB* dataset. We also show the number of CNs, denoted #CN. When Tmax is small, the number of CNs is small. As shown in Fig. 10(a), when the number of cores is 2, the speedups of all three algorithms are linear, but when the number of cores increases, the processing time of CLP and OLP keeps constant because when the number of operations is small, a few costly operations will become dominant, making the workload skewed among cores. DLP can solve the problem through dividing operations when the number of cores increases to 16. In Fig. 10(b) and Fig. 10(c), when Tmax increases, the number of CNs also increases sharply. The curves of all three algorithms become more smoother than Fig. 10(a) because more operations are involved. The DLP algorithm can get a near linear performance. The performance of CLP is worst because much redundant work will be done on each core when Tmax is large. The OLP algorithm performs in between. In Fig. 10(d), when Tmax is increased to 7,



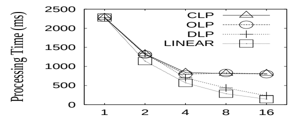
(a) Ksel = 1, $\#CN = 5,093$



(b) Ksel = 2, $\#CN = 5,093$



(c) Ksel = 4, $\#CN = 5,093$



(d) Ksel = 5, $\#CN = 5,093$

Figure 12: Vary Ksel (*IMDB*)

the number of *CN*s is increased to 26,642. Such large number of *CN*s makes the accumulated error very large for the CLP algorithm. Thus the performance of CLP is very bad. The DLP and OLP algorithms all perform good because the accumulated error is small and the workloads are well balanced among all cores.

Exp-2 (Vary Knum): The processing time for queries with different keyword numbers are shown in Fig. 11. We show the curves for Knum =2, Knum =3, Knum =5 and Knum =6 in Fig. 16(a), Fig. 16(b), Fig. 16(c), and Fig. 16(d) respectively. For Knum =4, the curve is exactly the same as Fig. 10(b) because Knum =4 is the default parameter for Knum, and Fig. 10(b) shows the curves when all parameters are set to their default values. Fig. 11(a) and Fig. 11(b) show that, when the number of keywords is small, the performances of CLP and OLP are bad because all operations are related to the few number of keywords, and thus the workloads of operations is hardly to be divided evenly. DLP improves the performance through data level dividing. In Fig. 11(c) and Fig. 11(d), when the number of keywords is large, operations are related to different keywords and their large number of combinations. It is easy for the tasks to be divided into different cores even in the *CN* level parallelism. As a result, the performance of all three algorithms are similar, and all are close to the linear performance.

Exp-3 (Vary Ksel): Keyword selectivity will not influence the number of *CN*s generated, but it influences the performances of all three algorithms because large keyword selectivity will generate more intermediate tuples. Fig. 12 shows the performance of all algorithms when varying Ksel from 1 to 5 in the *IMDB* dataset. The curves for Ksel =4 are also shown in Fig. 10(b) because 4 is the default value of Ksel. In all testings, the shapes for CLP and OLP does not change too much because when Ksel increases, the costs for all operations increase, and the total number of operations does not change, thus the processing times for all cores increase. The shape of the DLP algorithm is sensitive to Ksel, because when the costs of all operations increase, only part of the operation is allowed to be partitioned in DLP, thus the adaptive scheduling will change much. As a result, in Fig. 12(a) and Fig. 12(d), the performance of DLP is nearly linear, and in Fig. 12(b) and Fig. 12(c), the performance of DLP is close to the performance of CLP and OLP, but is still the best of all the three algorithms.

10. CONCLUSION

In this paper, we study the parallel keyword search problem on *RDB*s, which is a parallel multiquery optimization problem. We analyze three properties of the problem, regarding large number of SQLS, large sharing among SQLS and large number of intermediate tuples respectively. We propose three algorithms to solve the problem in different levels of parallelism. We analyze all algorithms and provide the time complexities. We conducted extensive experimental studies using two large real datasets, and confirmed the efficiency of our approaches.

Acknowledgment: The work was supported by grants of the Research Grants Council of the Hong Kong SAR, China No. 419008 and 419109.

11. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *Proc. of ICDE'02*, 2002.

[2] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: Authority-based keyword search in databases. In *Proc. of VLDB'04*, 2004.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *Proc. of ICDE'02*, 2002.

[4] D. Bozdag, F. Özgüner, and Ü. V. Çatalyürek. Compaction of schedules and a two-stage approach for duplication-based dag scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 20(6):857–871, 2009.

[5] S. Chaudhuri. An overview of query optimization in relational systems. In *Proc. of PODS'98*, 1998.

[6] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. *PVLDB*, 1(1), 2008.

[7] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *Proc. of ICDE'07*, 2007.

[8] S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. In *Networks*, 1972.

[9] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *Proc. of SIGMOD'08*, 2008.

[10] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2), 1993.

[11] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *Proc. of SIGMOD'07*, 2007.

[12] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-Style keyword search over relational databases. In *Proc. of VLDB'03*, 2003.

[13] V. Hristidis, H. Hwang, and Y. Papakonstantinou. Authority-based keyword search in databases. *ACM Trans. Database Syst.*, 33(1), 2008.

[14] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *Proc. of VLDB'02*, 2002.

[15] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proc. of VLDB'05*, 2005.

[16] A. Kementsietsidis, F. Neven, D. V. de Craen, and S. Vansummeren. Scalable multi-query optimization for exploratory queries over federated scientific databases. *PVLDB*, 1(1), 2008.

[17] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *Proc. of PODS'06*, 2006.

[18] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *Proc. of SIGMOD'08*, 2008.

[19] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *Proc. of SIGMOD'06*, 2006.

[20] H. Lu. *Query Processing in Parallel Relational Database Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[21] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *Proc. of SIGMOD'07*, 2007.

[22] A. Markowetz, Y. Yang, and D. Papadias. Keyword search on relational data streams. In *Proc. of SIGMOD'07*, 2007.

[23] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Comput. Surv.*, 24(1), 1992.

[24] S. Park, T. Kim, J. Park, J. Kim, and H. Im. Parallel skyline computation on multicore architectures. In *Proc. of ICDE'09*, pages 760–771, 2009.

[25] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: The power of rdbms. In *Proc. of SIGMOD'09*, 2009.

[26] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Querying communities in relational databases. In *Proc. of ICDE'09*, 2009.

[27] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Scalable keyword search on large data streams. In *Proc. of ICDE'09*, 2009.

[28] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proc. of SIGMOD'00*, 2000.

[29] S. Tatikonda and S. Parthasarathy. Mining tree-structured data on multicore systems. *PVLDB*, 2(1):694–705, 2009.

[30] C.-H. Yang, P. Lee, and Y.-C. Chung. Improving static task scheduling in heterogeneous and homogeneous computing systems. In *Proc. of ICPP'07*, 2007.

[31] C. Yu and W. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann, 1998.

[32] J. X. Yu, L. Qin, and L. Chang. *Keyword Search in Databases*. Morgan and Claypool Publishers, 2010.

[33] P. S. Yu, M. syan Chen, J. L. Wolf, and J. Turek. Parallel query processing. *ACM Computing Surveys*, 16:399–433, 1993.

[34] J. Zhou, P.-A. Larson, J. C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *Proc. of SIGMOD'07*, 2007.

APPENDIX

A. STATE OF ART

The left part of Fig. 13 shows the optimal plans for two *CN*s c_1 and c_2 in Fig. 2. The global optimal plan for evaluating the two *CN*s $c_1 \cup c_2$ together is also shown on the right part of Fig. 13. The subexpression $A\{k_1\} \bowtie W \bowtie P$ is part of the global optimal plan but not part of the optimal plan for c_2. Suppose the estimated costs for the optimal plans of c_1, c_2 and $c_1 \cup c_2$ are $cost(c_1)$, $cost(c_2)$ and $cost(c_1 \cup c_2)$ respectively. We have $cost(c_1 \cup c_2) \le cost(c_1) + cost(c_2)$.

B. *CN* LEVEL PARALLELISM

In Algorithm 1, we first sort all *CN*s in descending order of their costs so that *CN*s with large cost could be distributed first. In line 2-3, two structures *list* and *totalcost* are initialized. *list* keeps the set of *CN*s in each partition and *totalcost* maintains the total cost of each partition. In line 4-7, each *CN* is distributed to a certain partition, and each time, the selected *CN* is distributed to the partition with the least total cost/workload(line 5). Line 6-7 update *list* and *totalcost* after distributing the *CN* c_i. Finally, after the scheduling, *CN*s in each partition are processed on a certain core using the existing sequential algorithm(line 8-10).

In Algorithm 2, we use a max heap $\mathcal{H}$ to maintain the priorities of *CN*s not been distributed (line 2), and use *mincost* to keep the not-shared/extra cost for each *CN* which is initialized as the cost of the *CN* (line 3-6). We use G^i_E to denote the execute graph to be processed on core i, and use $G^i_{E'}$ to denote the execute graph not been processed on core i which is used for calculating *mincost* and is initialized on line 7-9. In line 11, we select a *CN* c_q with maximum extra cost from the heap $\mathcal{H}$ (line 11) and distribute it to a partition p with minimum cost after adding c_q. Line 13-14 update G^p_E and $G^p_{E'}$ by removing the execute plan graph of c_q from $G^p_{E'}$ to G^p_E. Line 15-17 update *mincost* and $\mathcal{H}$ according to the new cost left on $G^p_{E'}$ (the extra cost).

Proof sketch of Theorem 5.1: In Algorithm 1, sorting all *CN*s needs $O(m \cdot \log m)$ time. The loop in line 5-7 needs m iterations, and for each iteration, line 5 needs $O(n)$ time to find the minimum *totalcost* and line 6 needs $O(\mathsf{Tmax})$ time to add a *CN*. The total time complexity is $O(m \cdot \log m) + O(m \cdot (n + \mathsf{Tmax})) = O(m \cdot (n + \mathsf{Tmax} + \log m))$. □

Proof sketch of Theorem 5.2: In Algorithm 2, we only need to analyze the dominant part of the algorithm, which is line 12, line 15-17 inside the for loop of line 10. We have totally m loops, and in each loop, line 12 needs $O(n \cdot \mathsf{Tmax})$ time to find the best partition because we need to check all n partitions, and each time we need $O(\mathsf{Tmax})$ time to calculate the cost. In line 15-17, we should find all *CN*s that the extra cost in partition p is non-zero and at the same time overlaps the execute plan tree of c_q. The number of such *CN*s can be at most $|E(G_E)| \times n$ in all loops of i. This is because a *CN* overlaps c_q if and only if its execute plan tree contains an edge in G_E such that one end of the edge is a node in the execute plan tree of c_q. The number of such edges is at most $|E(G_E)| \times n$ since there are initially n execute graphs $G^i_{E'}(1 \le i \le n)$ and once the *CN* c_q is distributed, all such edges are removed from $G^p_{E'}$. Line 17 needs $O(\log m)$ to update the heap. The overall time complexity is $O(m \cdot n \cdot \mathsf{Tmax} + |E(G_E)| \cdot n \cdot \log m)$. Since $|E(G_E)| \le m \cdot \mathsf{Tmax}$, the overall time complexity becomes $O(m \cdot \log m \cdot \mathsf{Tmax} \cdot n)$. □

The change of $G^p_{E'}$: Fig. 14 illustrates the process of updating $G^1_{E'}$ and *mincost* after distributing c_7. The dashed lines are the edges to be removed and the *CN*s marked in circles are the *CN*s whose *mincost* should be updated. For example, for c_1, $mincost_1$ is reduced from 715 to 115 because a subexpression of cost 600 is removed.

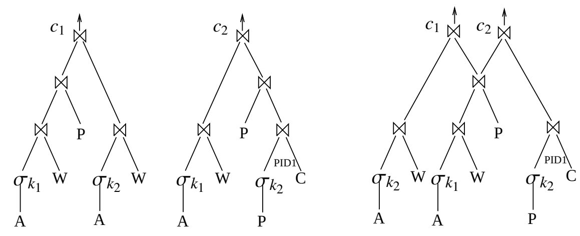


Figure 13: Optimal plans for c_1, c_2 and $c_1 \cup c_2$

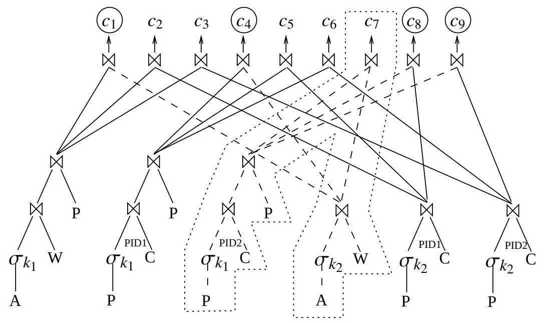


Figure 14: The change of $G^p_{E'}$

C. OPERATION LEVEL PARALLELISM

In Algorithm 3, the loop in line 4 processes from phase 1 to phase $depth(G_E)$. In each phase l, line 5-6 re-estimate the costs of all nodes in level l because all results of their child nodes has been calculated in previous phases, and thus we can use the new results in child nodes to get more accurate cost estimation as discussed in the paper. In Line 7, the procedure adaptive-distribute is for workload balancing, the nodes are distributed adaptively to cores that finish their current task, and each time, the node with the largest cost is distributed and evaluated, because all nodes in level l can be independently evaluated. The procedure one-level-estimate and adaptive-distribute are shown in line 8-14 and line 15-22 respectively and are self explained.

Proof sketch of Theorem 6.1: In Algorithm 3, there are at most Tmax phases and at most $\mathsf{Tmax} \times m$ nodes to be processed. In line 18, and line 21, the *enheap* and *deheap* operations in all phases need at most $O(\mathsf{Tmax}{\cdot}m{\cdot}\log(\mathsf{Tmax} \cdot m)) = O(\mathsf{Tmax}{\cdot}m{\cdot}(\log \mathsf{Tmax} + \log m))$ time. As a result, the time complexity of Algorithm 3 for scheduling is $O(\mathsf{Tmax} \cdot m \cdot (\log \mathsf{Tmax} + \log m))$. □

D. DATA LEVEL PARALLELISM

In Algorithm 4, we only show the extended part of Algorithm 3, which is used for adaptive data partitioning. As shown in line 3-10 of Algorithm 4, before processing each level l, we first check whether the most cost operation in the level has the potential to become the dominant cost, this is done by checking whether the most cost operation is larger than twice the n-th largest cost among all operations (line 4-5). If so, we should partition the operation into two operations. Since the most cost operation has not been processed, we should choose one of its child nodes to partition. The one with the minimum cost is chosen for partitioning (line 6). Line 7 invokes another procedure partition to add copies of the

Algorithm 1 CLP-Naive (C, n)

```
Input: the set of CNs $C = \{c_1, c_2, ..., c_m\}$, the number of cores $n$.
1: sort all CNs s.t. $cost(c_i) \geq cost(c_j)$ if $i \leq j$;
2: for $i = 1$ to $n$ do
3:     $list_i \leftarrow \emptyset$; $totalcost_i \leftarrow 0$;
4: for $i = 1$ to $m$ do
5:     $p \leftarrow argmin_{1 \leq j \leq n}(totalcost_j)$;
6:     $list_p \leftarrow list_p \cup \{c_i\}$;
7:     $totalcost_p \leftarrow totalcost_p + cost(c_i)$;
8: for $i = 1$ to $n$ in parallel do
9:     create execute graph $G_E^i$ for all CNs in $list_i$;
10:    process $G_E^i$ on core $i$;
```

Algorithm 2 CLP (C, n)

```
Input: the set of CNs $C = \{c_1, c_2, ..., c_m\}$, the number of cores $n$.
1: create the execute graph $G_E$ for all CNs in $C$;
2: max heap $\mathcal{H} \leftarrow \emptyset$;
3: for $i = 1$ to $m$ do
4:     $T_i \leftarrow$ the execute plan tree in $G_E$ for $c_i$;
5:     $mincost_i \leftarrow cost(T_i)$;
6:     $\mathcal{H}.enheap(i, mincost_i)$;
7: for $i = 1$ to $n$ do
8:     $G_E^i \leftarrow \emptyset$;
9:     $G_{E'}^i \leftarrow$ a copy of $G_E$;
10: for $i = 1$ to $m$ do
11:     $(q, mincost_q) \leftarrow \mathcal{H}.deheap()$;
12:     $p \leftarrow argmin_{1 \leq j \leq n}(cost(G_E^j \cup T_q))$;
13:     $G_E^p \leftarrow G_E^p \cup T_q$;
14:     $G_{E'}^p \leftarrow G_{E'}^p - T_q$;
15:     for all $T_j$ s.t. $T_j \cap T_q \neq \emptyset \wedge mincost_j > 0$ do
16:         $mincost_j = min\{mincost_j, cost(T_j \cap G_{E'}^p)\}$;
17:         $\mathcal{H}.update(j, mincost_j)$;
18: for $i = 1$ to $n$ in parallel do
19:     process $G_E^i$ on core $i$;
```

nodes to be partitioned in the execute graph as well as the copies of nodes who share them. Line 8 divides the relation of the child node to be partitioned uniformly into two halves and line 9 re-estimate the costs for all nodes in the current level, because after partitioning, the data of the child nodes for nodes in current level have changed, thus the costs for nodes in the current level should be changed accordingly. The partition procedure is shown in line 13-21. It is a recursive function. We first add a copy of the current node to be partitioned (line 13-14), then we recursively add copies of all its father nodes (line 15-16). After adding each father node, two types of edges should be added. The first type is an edge from the father node to the current node to be partitioned (line 17), and the second type is the edge from the new added father node to the child nodes of the original father node, unless the child node is the current node (line 18-20). This is because although the copied father node is new, its input nodes (except the current node) remain the same.

Proof sketch of Lemma 7.1: In each phase of Algorithm 4, we call the node that is not divided in the phase the original node and the node that is generated by dividing another node the copied node. Suppose now in the current phase, $n-1$ iterations of partitioning has finished. It is obvious that there are at least n copied nodes in the current phase. For each of the copied node u, we have $cost(Q[1]) \leq 2 \times cost(u)$. It is because (1) u is generated by partitioning a node w with maximum cost in one of previous iterations, thus $cost(w) \leq 2 \times cost(u)$, and (2) the value of $cost(Q[1])$ is non increasing because $Q[1]$ is partitioned in each iteration, thus $cost(Q[1]) \leq cost(w)$. Now we consider two situations, (1) $Q[n]$ is a copied node. In

Algorithm 3 OLP (C, n)

```
Input: the set of CNs $C = \{c_1, c_2, ..., c_m\}$, the number of cores $n$.
1: create the execute graph $G_E$ for all CNs in $C$;
2: for $i = 1$ to $n$ do
3:     $G_E^i \leftarrow emptyset$;
4: for $l = 1$ to $depth(G_E)$ do
5:     for all node $v$ in level $l$ do
6:         one-level-estimate $(v)$;
7:     adaptive-distribute $(l)$;

8: Procedure one-level-estimate $(v)$
9: if $v$ is a selection then
10:    suppose $v = R\{K\}$;
11:    $cost(v) \leftarrow |R| \times \Pi_{k \in K} sel(k)$;
12: else
13:    suppose $v = R.a_i \bowtie R'.a_j$ and $a_i \rightarrow a_j$;
14:    $cost(v) \leftarrow |R| \times |R'| \times sel(a_i \rightarrow a_j)$;

15: Procedure adaptive-distribute $(l)$
16: max heap $\mathcal{H}_l \leftarrow \emptyset$;
17: for all nodes $v$ in level $l$ of $G_E$ do
18:     $\mathcal{H}_l.enheap(v, cost_v)$;
19: for $i = 1$ to $n$ in parallel do
20:     while $\mathcal{H}_l \neq \emptyset$ do
21:         $(v, cost_v) \leftarrow \mathcal{H}_l.deheap()$;
22:         process $v$ on core $i$;
```

such condition, from the above analysis, we have $cost(Q[1]) \leq 2 \times cost(u)$. (2) $Q[n]$ is a original node. Since there are n nodes in Q and at least n copied nodes in total, there is a node u, such that $u \notin Q$ and u is a copied node. We have $cost(Q[1]) \leq 2 \times cost(u)$ and $cost(u) \leq cost(Q[1])$, thus $cost(Q[1]) \leq 2 \times cost(u)$. In both situations, we can all conclude that after $n-1$ partitioning, we have $cost(Q[1]) \leq 2 \times cost(u)$, so at most $n-1$ partition operations will be performed. □

Proof sketch of Theorem 7.2: We first prove that in each level, each partition operation for a node will generate at most $\mathsf{Tmax} \times n$ copies in the final execute graph. Since partitioning a node in a certain level will add at most one copy for each node who share it in higher levels, and from Lemma 7.1, we know that at most $depth(G_E) \times n \leq \mathsf{Tmax} \times n$ partition operations will be performed. Each node will have at most $\mathsf{Tmax} \times n$ copies. Since Algorithm 3 has time complexity $O(\mathsf{Tmax} \cdot m \cdot (\log \mathsf{Tmax} + \log m))$, and m can be changed to $m \times \mathsf{Tmax} \times n$ in the worst case, the time complexity is changed to $O(\mathsf{Tmax}^2 \cdot m \cdot n \cdot (\log n + \log \mathsf{Tmax} + \log(m \cdot \mathsf{Tmax} \cdot n))) = O(\mathsf{Tmax}^2 \cdot m \cdot n \cdot (\log n + \log \mathsf{Tmax} + \log m))$. □

Proof sketch of Theorem 7.1: We first prove that after partitioning, the total cost remains the same. When A is partitioned into $A = A_1 \cup A_2$ where $A_1 \cap A_2 = \emptyset$. $A \bowtie B = (A_1 \cup A_2) \bowtie B = (A_1 \bowtie B) \cup (A_2 \bowtie B)$ where $(A_1 \bowtie B) \cap (A_2 \bowtie B) = (A_1 \cap A_2) \bowtie B = \emptyset$. We have $cost(A \bowtie B) = cost((A_1 \bowtie B) \cup (A_2 \bowtie B)) = cost(A_1 \bowtie B) + cost(A_2 \bowtie B) - cost((A_1 \bowtie B) \cap (A_2 \bowtie B)) = cost(A_1 \bowtie B) + cost(A_2 \bowtie B)$. It means that the total cost remain unchanged after partitioning A into A_1 and A_2. We now give a lower bound of the cost for the optimal solution. Since the total cost of all operations is $\sum_{v \in V(G_E)} cost(v)$, it is obvious that the cost of any solution can not be smaller than $\frac{\sum_{v \in V(G_E)} cost(v)}{n}$. We use $\underline{cost} = \frac{\sum_{v \in V(G_E)} cost(v)}{n}$ to denote a cost lower bound of the optimal solution, use $cost$ to denote the cost of our solution, and use $cost^*$ to denote the cost of the optimal solution. We have $\underline{cost} \leq cost^*$.

Let $V_i(G_E)$ be all nodes in level i of G_E. We then prove that in any phase l, the maximum cost for cores is no larger than $2 \times \frac{\sum_{v \in V_i(G_E)} cost(v)}{n}$, i.e., $max_{1 \leq j \leq n}(totalcost_{l,j}) \leq 2 \times \frac{\sum_{v \in V_i(G_E)} cost(v)}{n}$. Suppose the $totalcost$ values are ranked in non-increasing order, i.e.,

Algorithm 4 DLP (C, n)

```
Input: the set of CNs $C = \{c_1, c_2, ..., c_m\}$, the number of cores $n$.
 1: create the execute graph $G_E$ and initialize $G_E^i$ for $1 \le i \le n$;
 2: for $l = 1$ to $depth(G_E)$ do
 3:     repeat
 4:         $Q \leftarrow$ top-$n$ nodes in level $l$ with maximum cost in descending
            order of cost;
 5:         if $cost(Q[1]) > 2 \times cost(Q[n])$ then
 6:             $v \leftarrow argmin_{u \in cnodes(Q[1])}(cost(u))$;
 7:             $v' \leftarrow$ partition ($v$);
 8:             divide tuples in $v$ into two halves uniformly and put one half
                into $v'$;
 9:         one-level-estimate ($u$) for all node $u$ in level $l$;
10:     until $cost(Q[1]) \le 2 \times cost(Q[n])$
11:     adaptive-distribute ($l$)

12: Procedure partition ($v$)
13: $v' \leftarrow$ a copy of $v$;
14: $V(G_E) \leftarrow V(G_E) \cup \{v'\}$;
15: for all $v_f \in fnodes(v)$ do
16:     $v'_f \leftarrow$ partition ($v_f$);
17:     $E(G_E) \leftarrow E(G_E) \cup \{(v'_f, v')\}$;
18:     for all $v_c \in cnodes(v_f)$ do
19:         if $v_c \ne v$ then
20:             $E(G_E) \leftarrow E(G_E) \cup \{(v'_f, v_c)\}$;
21: return $v'$;
```

for any $1 \le i \le j \le n$, $totalcost_{l,i} \ge totalcost_{l,j}$. We only need to prove $totalcost_{l,1} \le 2 \times totalcost_{l,n}$, because $totalcost_{l,n} \le \frac{\sum_{v \in V_i(G_E)} cost(v)}{n}$. Since the *totalcost* is created by adding the most cost node in level l iteratively, we prove it by induction on the number of iterations. We choose the initial point to be the n-th iteration, in such situation, $totalcost_{l,i} = cost(Q[i])$ for all $1 \le i \le n$. Since $cost(Q[1]) \le 2 \times cost(Q[n])$, we have $totalcost_{l,1} \le 2 \times totalcost_{l,n}$ holds. Now suppose in iteration $i (i \ge n)$, $totalcost_{l,1} \le 2 \times totalcost_{l,n}$ holds, we prove in iteration $i + 1$, after adding a value $cost(v)$ on $totalcost_{l,n}$ and resorting the *totalcost* values in non-increasing order, $totalcost_{l,1} \le 2 \times totalcost_{l,n}$ also holds. When $n \le 2$, it is straightforward. Now suppose $n \ge 3$. For ease of reference, we use $totalcost^i$ to denote the *totalcost* values after the i-th iteration. There are two situations, (1) after adding $cost(v)$, $totalcost^{i+1}_{l,n} = totalcost^i_{l,n} + cost(v)$ is still the smallest value. We have $totalcost^{i+1}_{l,1} = totalcost^i_{l,1} \le 2 \times totalcost^i_{l,n} \le 2 \times (totalcost^i_{l,n} + cost(v)) = 2 \times totalcost^{i+1}_{l,n}$, and (2) after adding $cost(v)$, $totalcost^i_{l,n} + cost(v)$ is not the smallest value. We have $totalcost^{i+1}_{l,n} = totalcost^i_{l,n-1}$. As a result, $totalcost^{i+1}_{l,1} = max\{totalcost^i_{l,1}, totalcost^i_{l,n} + cost(v)\} \le 2 \times totalcost^i_{l,n} \le 2 \times totalcost^i_{l,n-1} = 2 \times totalcost^{i+1}_{l,n}$. In both cases, $totalcost^{i+1}_{l,1} \le 2 \times totalcost^{i+1}_{l,n}$ holds.

From the above analysis, we have the following result: $cost \le \sum_{l=1}^{depth(G_E)} (max_{1 \le j \le n}(totalcost_{l,j})) \le \sum_{l=1}^{depth(G_E)} (2 \times \frac{\sum_{v \in V_i(G_E)} cost(v)}{n}) = 2 \times \frac{\sum_{v \in V(G_E)} cost(v)}{n} = 2 \times \underline{cost} \le 2 \times cost^*$, and thus, Theorem 7.1 holds. □

Some Remarks of Theorem 7.2 and Theorem 7.1: The time complexity of Algorithm 4 is $O(\mathsf{Tmax}^2 \cdot m \cdot n \cdot (\log n + \log \mathsf{Tmax} + \log m))$, as shown in Theorem 7.2. Since for our problem, we assume $m >> n$ and $m >> \mathsf{Tmax}$ in general, the time complexity is still much smaller than $O(m^2)$. Theorem 7.1 shows that Algorithm 4 is a 2-approximate algorithm comparing to the optimal solution on the divided execute graph. Such approximate ratio is made under the assumption that all cost functions used is accurate. Note that such approximate ratio dose not hold for Algorithm 3 performed on the original execute graph G_E.

Ksel	Keywords
1	child rose live secretary
2	singer adventure police mark
3	Jack Peter romance family nurse reporter
4	guest officer presenter Michael
5	John show various comedy

Table 1: Keywords used for the *IMDB* dataset

Ksel	Keywords
1	theory impact server similarity
2	method level minimum block
3	network rate resource rule task sharing
4	analysis management retrieval single
5	system performance multiple technique

Table 2: Keywords used for the *DBLP* dataset

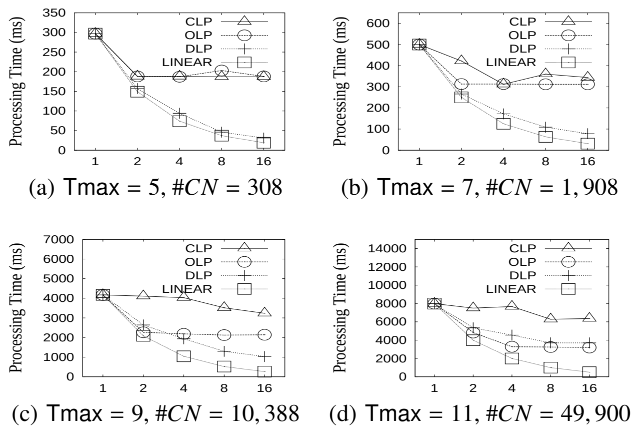
E. ADDITIONAL RELATED WORK

In the literature, for a keyword query on a relational database, it returns a set of inter-connected structures in the *RDB* that contain the user given keywords. The techniques to answer keyword queries in *RDB*s are mainly in two categories: *CN*-based (schema-based) and graph based (schema-free) approaches.

In the *CN*-based approaches [1, 14, 12, 21, 22, 25, 27], it processes a keyword query in two steps, namely, candidate network (*CN*) generation and *CN* evaluation. *DBXplorer* [1], Discover [14] and *KRDBMS* [25] focused on retrieving connected trees using SQL on RDBMSs. In [22], Markowetz et al. discussed how to efficiently generate all *CN*s and how to process keyword queries in an *RDB* stream environment based on a sliding window model. Among these approaches, in *DISCOVER-II* [12], Hristidis et al. incorporated IR-style ranking techniques to rank the connected trees. In *SPARK* [21], Luo et al. proposed a new ranking function by treating each connected tree as a virtual document, and unified the AND/OR semantics in the score function using a parameter. The ranking issues were also discussed in [2, 13, 19].

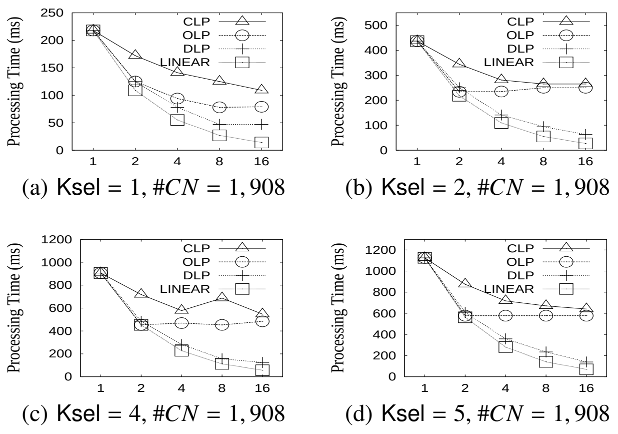
Finding top-k interconnected structures has been extensively studied in graph based approaches in which an *RDB* is materialized as a weighted database graph. The representative works on finding top-k connected trees are [3, 15, 17, 7, 9]. In brief, finding the exact top-k connected-trees is an instance of the group Steiner tree problem [8], which is NP-hard. To find top-k connected trees, Bhalotia et al. proposed backward search in *BANKS-I* [3], and Kacholia et al. proposed bidirectional search in *BANKS-II* [15]. Kimelfeld et al. [17] proposed a general framework to retrieval top-k connected trees with polynomial delay under data complexity, which is independent of the underline minimum Steiner tree algorithm. Ding et al. in [7] also introduced a dynamic programming approach to find the minimum connected tree and approximate top-k connected trees. Golenberg et al. in [9] attempted to find an approximate result in polynomial time under the query and data complexity.

Top-k connected trees are hard to compute, in *BLINKS* [11], He et al. proposed the distinct root semantics. In *BLINKS*, search strategies were proposed with a bi-level index built to fast compute the shortest distances. Dalvi et al. [6] conducted keyword search on external memory graphs under the distinct root semantics. Li et al. in *EASE* [18] defined an r-radius Steiner graph, where each r-radius Steiner graph is a subpart of a maximal r-radius subgraph. Qin et al. studied multi-center communities under the distinct core semantics in [26], and proposed new polynomial delay algorithms to compute all or top-k communities.



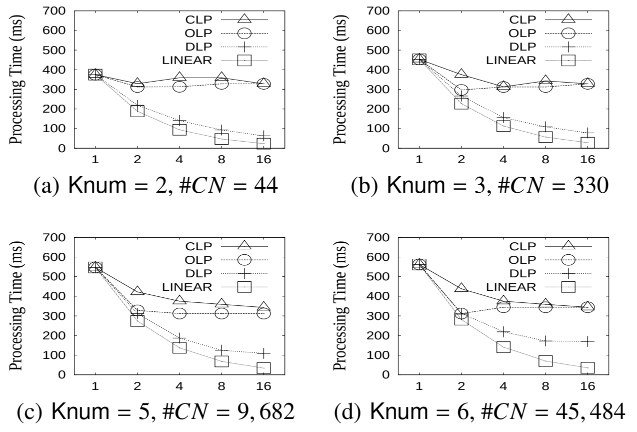
(a) Tmax = 5, $\#CN = 308$ (b) Tmax = 7, $\#CN = 1,908$ (c) Tmax = 9, $\#CN = 10,388$ (d) Tmax = 11, $\#CN = 49,900$

Figure 15: Vary Tmax (*DBLP*)



(a) Ksel = 1, $\#CN = 1,908$ (b) Ksel = 2, $\#CN = 1,908$ (c) Ksel = 4, $\#CN = 1,908$ (d) Ksel = 5, $\#CN = 1,908$

Figure 17: Vary Ksel (*DBLP*)



(a) Knum = 2, $\#CN = 44$ (b) Knum = 3, $\#CN = 330$ (c) Knum = 5, $\#CN = 9,682$ (d) Knum = 6, $\#CN = 45,484$

Figure 16: Vary Knum (*DBLP*)

F. PERFORMANCE STUDIES

The *DBLP* schema includes the following four relations: Paper(Paperid, Title), Author(Authorid, Authorname), Write(Writeid, Authorid, Paperid) and Cite(Citeid, Paperid1, Paperid2). The primary key for each relation is underlined. The size of the raw data for the *DBLP* dataset is 686MB. The number of tuples for the four relations are 1,341,055, 789,586, 3,419,237, and 112,387 respectively, and the total number of tuples in *DBLP* is 5,662,265.

For all testings, we vary 3 parameters, namely, the average keyword selectivity Ksel, the keyword number Knum, and the size control parameter Tmax. Every parameter has a default value. For *DBLP*, the settings are similar to *IMDB*. The general keywords selected with different keyword selectivity are shown in Table 2 and by default, the keyword selectivity Ksel is 3. The keyword number Knum ranges from 2 to 6 with a default value 4 and the size control parameter Tmax ranges from 5 to 11 with a default value 7.

Exp-1 (Vary Tmax**):** The curves for testings in the *DBLP* dataset when vary Tmax are shown in Fig. 15. Fig. 15(a) and 15(b) show that, when Tmax is small, the CLP and OLP algorithms performs bad because the workload skew is the main bottleneck when Tmax is small. DLP performs near linear because it can divide a large operation near evenly into different cores. Fig. 15(c) shows that when Tmax is increased to 9, the performance of OLP is much better than CLP. This is because the bottleneck is changed from the workload skew problem to the large inter-core sharing problem for the CLP algorithm, while OLP can solve such problem by processing the same *CN* using different cores. In Fig. 15(d), OLP and DLP have similar performances and both are two times more faster than CLP when the number of cores is larger than 2. OLP is faster than DLP in some cases because DLP needs more time dividing relations.

Exp-2 (Vary Knum**):** Fig. 16 shows the experimental results when varying the number of keywords in the *DBLP* dataset. In Fig. 16(a) and Fig. 16(b), when Knum is small, both CLP and OLP have bad performance because the number of *CN*s is small, and thus the probability of workload skew is high. DLP performs near ideally in such cases because it is easier for a few costly operations to be divided evenly. Fig. 16(c) and Fig. 16(d) illustrate the cases when Knum is large. The curves in Fig. 16(c) and Fig. 16(d) are much different from the corresponding cases in the *IMDB* dataset in Fig. 11(c) and Fig. 11(d) respectively. This is because the *IMDB* schema is more complex than the *DBLP* schema. In *DBLP*, there are only 4 relations and 2 of them can contain keywords while in *IMDB*, there are 8 relations and 7 of them can contain keywords. Generally speaking, for CLP and OLP, the more complex the schema is, the higher probability the tasks can be divided evenly. In *DBLP*, even when the number of keywords is large, the workload skew problem is still the bottleneck because of the simple schema used. In Fig. 16(d), when the number of cores increases, the processing time for OLP increases in some cases. It is because when workload skew exists, OLP needs more time rescheduling when the number of cores is larger. DLP performs best in all cases.

Exp-3 (Vary Ksel**):** The experimental results for queries with different keyword selectivity in the *DBLP* dataset are shown in Fig. 17. In Fig. 17(a), when the keyword selectivity is small, the gap between DLP and OLP is small, and in Fig. 17(b), when the keyword selectivity increases, the gap between DLP and OLP increases. This is because when the number of *CN*s keeps unchanged, the smaller the keyword selectivity is, the more sensitive the error of the cost estimation is. DLP can decrease the sensitivity of estimation error through dividing relations. In Fig. 17(c) and Fig. 17(d), the curves for all three algorithms do not differ too much. This is because when the keyword selectivity becomes large, the sensitivity of estimation error does not change too much.