Towards Real-time Parallel Processing of Spatial Queries *

Haibo Hu Manli Zhu Dik-Lun Lee
Department of Computer Science
The Hong Kong University of Science and Technology
{haibo, cszhuml, dlee}@cs.ust.hk

Abstract

Spatial databases are entering an era of mass deployment in various real-life applications, especially mobile and location-based services. The real-time processing of spatial queries to meet different performance goals poses new problems to the real-time and parallel processing communities. In this paper, we investigate how multiple window queries can be parallelized, decomposed, scheduled and processed in realtime workloads to optimize system performance, such as I/O cost, response time and miss rate. We devise in-memory R-trees to decompose queries into independent jobs. Jobs from different queries can be combined according to their spatial locality to eliminate redundant I/Os. Runtime job schedulers are elaborately devised to optimize response time or miss rate for various systems. Empirical results show a significant performance improvement over the sequential, unparalleled approach.

1 Introduction

Spatial databases (SDBs) are designed to handle large volume of spatial data. As mobile and ubiquitous computing becomes more and more important, the number of spatial applications is increasing. Many of them are location-based services (LBS) supporting mobile clients and dynamic queries whose results are subject to change according to the user's context, especially his location. For example, the result of a "nearest restaurant" query for an automobile user may become invalid after just one minute. Therefore, it is desirable for SDBs to process queries in real-time.

Traditional spatial database research focuses on optimizing I/O cost for a single query in centralized or paralleled environments. However, in real-time spatial database applications, especially location-based services, queries arrive at SDB in a stream. They often share common result objects due to locality. Therefore, inter-query optimization can reduce I/O cost and response time. On the other hand, a general real-time database system should adapt to various workloads to meet different system goals, e.g., minimizing system response time, execution cost or query miss rate. To achieve this objective, a dynamic scheduler that takes advantage of queries' spatial properties is needed. These considerations, stemmed from both real-time database and spatial database research, lead to our study on realtime query processing techniques for spatial databases.

In this paper, we investigate how multiple spatial queries (specifically, window queries) can be parallelized, decomposed, scheduled and processed under a realtime workload in order to enhance runtime performance, e.g., I/O cost, response time and miss rate. We take advantage of query locality to decompose and group overlapping queries into independent jobs. Jobs from different queries are combined so that redundant I/Os to retrieve the same objects can be minimized. Further, we design dynamic job schedulers to optimize system performance metrics (response time or miss rate) for different result returning modes. Empirical results from both synthetic data and real datasets show significant improvements on all performance metrics over a sequential processing approach. Although the performance gain is at the cost of additional computation and memory storage, we show that the overhead is relative small.

The remainder of this paper is organized as follows. Section 2 introduces some spatial database preliminaries and reviews related work on real-time query scheduling and spatial query optimization. The real-life SDB system model is presented in Section 3, followed by our proposed real-time query decomposing and job scheduling techniques in Section 4. Section 5 further derives the detailed scheduling policies. Empirical results are analyzed in Section 6, and finally the paper is concluded with future work.

2 Preliminary and Related Work

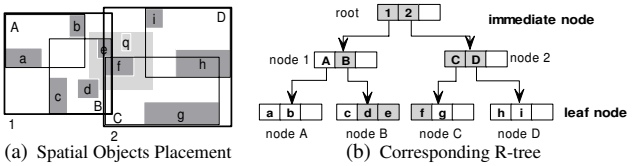
2.1 R-tree and Spatial Data Index

The predominant access method for spatial database is R-tree [5] and its variations. Many commercial database products, such as Oracle9i adopt R-tree to index spatial and geometric features of datasets. The R-tree is a direct ex-

*Supported by Research Grants Council, Hong Kong SAR under grant HKUST6079/01E and HKUST6225/02E.

Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03)
0190-3918/03 $ 17.00 © 2003 IEEE

tension of B-tree for multidimensional data. It is a balanced tree that consists of intermediate and leaf nodes. The MBRs (Minimal Bounding Rectangles) of the actual data objects are stored in the leaf nodes, and intermediate nodes are generated by grouping MBRs of its children nodes. Figure 1(a) illustrates the placement of spatial objects $a, b, ..., i$ and Figure 1(b) shows the corresponding R-tree, where $root$, $node\,1$ and 2 are intermediate nodes and the rest are leaf nodes. Each node in the R-tree has entries for its children in the form of (MBR, pointer), where MBR is the minimal bounding box of all objects in that child node and $pointer$ is the address of the child node. For leaf nodes, $pointer$ points to the actual data objects. For example, leaf node A has two pointers pointing to object a and b, respectively.



(a) Spatial Objects Placement (b) Corresponding R-tree

Figure 1. An Example of R-tree

2.2 Window Query Processing

Window (range) queries are one of the most common type of queries in spatial databases [12]. A *window query* requests for a set of objects that intersect a window q. The processing of a window query (e.g., the light gray window q in Figure 1(a)) in R-trees is as follows: starting from the root node, recursively search downwards for children whose MBRs overlap window q (designated by the gray nodes in Figure 1(b)). Among the leaf nodes that are searched, return those spatial data objects that overlap with window q. In the example shown in Figure 1, object d and e are returned when node B is searched, and f is returned when C is searched. But nothing is returned when D is searched.

2.3 Related Work

Much attention has been paid to the manipulation and scheduling of incoming queries in a real-time database system (RTDBS). Pang et al. proposed algorithms to minimize the number of missed deadlines by adapting both the multiprogramming level and the memory allocation strategy of an RTDBS through feedback [8]. In their subsequent work [7], they incorporated the notion of multiclass and devised dynamic algorithms that can ensure any deadline misses are scattered across the different classes according to a user-defined miss distribution. Their system adapts itself to current resource configuration and workload characteristics by tuning the admission and resource allocation policies. Garofalakis and Ioannidis modeled the full complexity of scheduling distributed multi-dimensional resource units in hierarchical parallel systems for intra- and inter-query executions [4]. They provided various heuristics for different scenarios of parallelism. However, while achieving good performance for general-purpose relational databases, these approaches are less efficient, or even inapplicable, to spatial databases since they don't take advantage of any spatial semantics of the queries.

On the other hand, several papers addressed the problem of spatial query optimization. Aref and Samet proposed a spatial database architecture called *SAND* where spatial and non-spatial components of an object are stored separately [1]. But the query processing and optimization algorithms incorporate both kinds of components to achieve better performance. In [9], Papadopoulos and Manolopoulos studied the performance of nearest neighbor queries in multi-disk multi-processor parallel architectures. They utilized statistical information to estimate the number of leaf-node accesses and to determine an efficient parallel execution strategy over all processors. However, these papers focus on single-query optimization and are offline algorithms. To the best of our knowledge, no previous work has addressed the problem of query scheduling and optimization in a real-time, online spatial database system.

3 Real-life Spatial Database System Model

System Architecture: A real-life spatial database system is commonly deployed in a distributed environment: clients (especially mobile clients) are the end users. Their spatial queries are either directly sent to SDB via a wired network, or through the mobile support stations (MSS) of a wireless network. Figure 2 illustrates the system model.

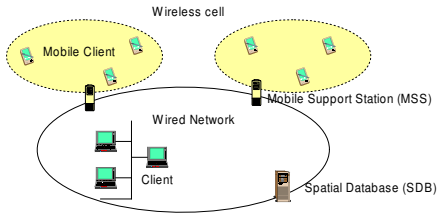


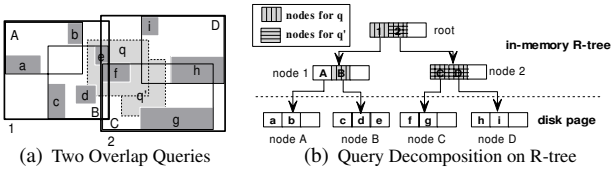
Figure 2. Distributed System Architecture

Performance Objectives: Our primary goal in this paper is to optimize the following performance metrics: **I/O cost**, **mean response time**, **miss rate**. These system metrics are not fully compatible to each other: minimizing the number of queries that missed the deadlines (called *overdue queries*) or average I/O cost may increase the overall system response time. In this paper, our strategy is to minimize average response time for systems not supporting deadlines, and to minimize miss rate for systems supporting deadlines. I/O cost is minimized in a best-effort fashion, which is discussed in the next section.

4 Processing Real-time Spatial Queries

The essential idea of real-time spatial query processing is to process multiple window queries in parallel in order

Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03)
0190-3918/03 $ 17.00 © 2003 IEEE



(a) Two Overlap Queries (b) Query Decomposition on R-tree

Figure 3. In-memory R-tree and Query Decomposition

to eliminate duplicate I/O accesses to common index and data pages. Meanwhile, jobs are re-scheduled to minimize mean query response time. In principle, processing queries close to one another will save more I/O cost because there is a high chance that they can share some MBRs in the R-tree index as well as data objects. However, how to quantify closeness and degree of overlapping in terms of I/O is rather difficult. In the following section, we propose our innovative solution based on window query decomposition.

4.1 In-memory R-tree and Active Queries

In a practical implementation where a fair amount of main memory is available, high-level R-tree nodes can be assumed to be cached in memory. In the extreme, the entire R-tree except the leaf nodes can be assumed to be cached in memory, because the total number of leaf nodes is normally two orders of magnitude larger than that of all higher level nodes. We call the cached part of the R-tree *in-memory R-tree*. Figure 3(b) illustrates the two parts of an R-tree. In general, such a two-way partitioning can be applied to an R-tree at any level. It depends on how much memory can be allocated for the in-memory R-tree. In the sequel, without loss of generality, the in-memory R-tree includes all of the internal nodes of an R-tree.

In terms of I/O workload, it is natural for us to divide the processing of a window query q into two stages:
Stage 1: perform the window query on the in-memory R-tree to identify all of the leaf nodes to visit, e.g., in Figure 3, when query q goes through this stage, leaf node B, C and D are identified;
Stage 2: for each of the leaf nodes, visit the corresponding page and retrieve resultant data objects. Such processing of a leaf node is called a *leaf-node job*. For example, B, C and D are leaf node jobs for query q.

It's obvious that Stage 1 is fast as it only incurs memory operations while Stage 2 is much more costly due to I/O operations. Multiple real-time window queries can quickly go through the first stage and queue before the second stage, where they, called *active queries*, are decomposed into leaf node jobs. We propose to put a *runtime scheduler* between the two stages for scheduling the leaf-node jobs in the queue.

The rationale of eliminating redundant I/Os lies in that leaf node jobs from different active queries may refer to the same leaf node. We intentionally combine those jobs referring to the same leaf node into one "super" leaf-node job (super job, for short). When executing this super job, all of its associated window queries will be processed altogether. For example, in Figure 3, query q has jobs B, C, D, while q' has jobs C, D. Thus leaf nodes C and D are required by both q and q'. We combine the two C jobs into one super C job. When executing super job C, q and q' are both processed at the same time. All objects intersecting either q or q' (i.e., f) will be returned. The same applies to D. Therefore all active queries accessing the same leaf node MBR require only one access to the leaf-node page. As such, the same object required by multiple queries is only retrieved once. For example, object f, a common result for q and q', is retrieved only once when executing super job C. In the sequel, we develop schedulers to determine the processing order of these super jobs.

4.2 The Scheduler

The scheduler in our proposal maintains an *active query table* holding a maximum of S queries. For each active query, it stores the information required for scheduling. The scheduler determines an optimal processing order of the super jobs, which is to be discussed in Section 5. If the table is full, new queries have to wait before Stage 1. After an active query is completed, that is, all its associated super jobs are executed, its entry is removed from the table. A new query is decomposed and its entry is inserted into the table.

4.3 Returning the Query Result

Different applications require different result returning modes, our system supports the two of them. In **immediate mode**, when one super job is finished, SDB immediately connects to the relevant clients and sends the partial result back. While in **collecting mode**, the scheduler in addition maintains a collector for each active query which stores so-far retrieved objects in memory. When the query is completed, the whole set of results is returned and the collector is removed.

4.4 Spatial Database System Architecture

To sum up this section, our spatial database system is composed of the following components and data structures: *query receiver* (with query registry storing clients and queries associations), *query decomposer* (with in-memory R-tree), the *scheduler* (with active query table), *super job executer* (with leaf pages of R-tree and data objects), *collectors* and the *transmitter*. Figure 4 illustrates the architecture. The scheduler is the core component, which is further discussed in the next section. Generally, it evaluates the active query table to schedule the next super job, which is executed by the super job executer. The resultant objects are directly sent to the transmitter (in immediate mode) or the collectors (in collecting mode). If a query is completed, the scheduler removes its entry from the active query table and request

Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03)
0190-3918/03 $ 17.00 © 2003 IEEE

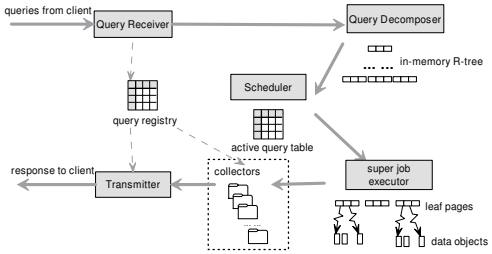


Figure 4. Spatial Database System Architecture

the query decomposer to decompose the next queued query in the query registry. The entry of this new query is then inserted into the active query table. The scheduler repeats the routine continually.

5 Scheduling Policies

In the active query table, super jobs and queries form a many-to-many mapping: a query is associated with multiple jobs through decomposition, while a super job is associated with multiple queries that comprise it. In this section, we devise super jobs scheduling policies for the scheduler to minimize the mean response time. The following assumptions and notations need to be introduced:

1. The object MBRs have similar sizes and are uniformly distributed in the MBR of the leaf node. Therefore, the number of resultant objects for query q in leaf page p, denoted by $selec(p,q)$, is proportional to the size of the overlapping area of q and MBR of p. This means, $selec(p,q) = fanout(p) \times \frac{overlap\_area(p,q)}{area(p)}$, where $fanout(p)$ is the total number of objects in leaf node p. Similarly, $selec(q)$ denotes the total number of resultant objects for query q, i.e., $selec(q) = \sum_{i=1}^{T} selec(p_i,q)$.[1] And $selec(p)$ denotes the total number of resultant objects for all active queries in super job p, i.e., $selec(p) = fanout(p) \times \frac{overlap\_area(p,q_1,q_2,\ldots,q_S)}{area(p)}$;[2]

2. Each object has equal data size. That is to say, the time cost for a job p is proportional to $selec(p)$. In the sequel, we exclusively use the latter measurement to derive the mean response time formulae.

5.1 Cost Model for Immediate mode

In the immediate mode, we define the response time for a query q, denoted by $resp\_time(q)$, as the average return time for each of the resultant object, i.e.,

$$resp\_time(q) = \frac{\sum_{j=1}^{selec(q)} [T_{q,j} - T_q]}{selec(q)} \quad (1)$$

where $T_{q,j}$ is the time when the jth object of q is returned, T_q is the time at which q is inserted in the active query table.[3] The mean response time for all queries in the active query table is,

$$\overline{resp\_time} = \frac{1}{S}\sum_{i=1}^{S}\frac{1}{selec(q_i)}\sum_{j=1}^{selec(q_i)}[T_{q_i,j} - T_{q_i}] \quad (2)$$

Since T_{q_i} and S are constant, we only need to minimize the following:

$$\sum_{i=1}^{S}\frac{1}{selec(q_i)}\sum_{j=1}^{selec(q_i)} T_{q_i,j} = \sum_{p=1}^{T}(t_p \cdot \sum_{i=1}^{S}\frac{selec(p,q_i)}{selec(q_i)}) \quad (3)$$

Equation 3 rewrites the summation of response time of all resultant objects in terms of the jobs in which they are retrieved. Here t_p is the finishing time of job p, which is the finishing time of its previous job p-1 plus $selec(p)$, the duration of job p, i.e., $t_p = t_{p-1} + selec(p)$.

$\frac{selec(p,q_i)}{selec(q_i)}$ represents the proportion of job p satisfying query q_i. It is denoted by $sat(p,q_i)$ in the sequel. Its summation over all q_i, denoted by $sat(p)$, represents the contribution of job p to all active queries, i.e., $sat(p) = \sum_{i=1}^{S} sat(p,q_i)$. Thus, Equation 3 is rewritten as follows:

$$\sum_{i=1}^{S}\frac{1}{selec(q_i)}\sum_{j=1}^{selec(q_i)} T_{q_i,j} = \sum_{p=1}^{T} t_p \cdot sat(p) \quad (4)$$

From Equation 4, the minimization problem is transformed into the following scheduling problem:

Problem 1 *Given a set of jobs p_1, p_2, $\cdots$, p_T, which take $selec(p_1)$, $selec(p_2)$, $\cdots$, $selec(p_T)$ time to finish, respectively. Each job will have a penalty proportional to the time it finishes, i.e., $penalty(p) = t_p \cdot w_p$. Find the optimal scheduling sequence to minimize the overall penalty $\sum_{p=1}^{T} t_p \cdot w_p$.*

This is the same problem as minimizing total weighted completion time of a sequence of jobs in a single processor, which was studied in [10] by Smith. He proved that scheduling these jobs in non-increasing order of their $w_p/selec(p)$ values produces the optimal solution. Therefore, our scheduling policy in immediate mode is:

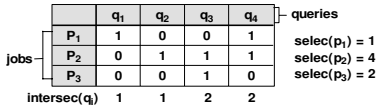
Scheduling Policy 1 *Immediate Mode Scheduling Policy: Arrange the jobs in descending order of their $sat(p)/selec(p)$ values.*

[1] T is the total number of jobs in the active query table.

[2] $overlap\_area(p,q_1,q_2,\ldots,q_S) = \cup_{i=1}^{S} overlap\_area(p,q_i)$, where S is the maximum number of queries the active query table can hold. In the implementation, this is approximated by the Monte Carlo randomized algorithm.

[3] The actual response time for a query q should also include the time q is queued before stage 2. However, as this period of time is only dependent on system throughput, which is regardless of scheduling policies, it's eliminated from subsequent analysis.

Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03)
0190-3918/03 $ 17.00 © 2003 IEEE



	q_1	q_2	q_3	q_4
P_1	1	0	0	1
P_2	0	1	1	1
P_3	0	0	1	0
intersec(q_i)	1	1	2	2

Figure 5. Job and Query Association

5.2 Cost Model for Collecting Mode

In the collecting mode, we define the response time of a query q, denoted by $resp-time(q)$, as the return time of the last obtained resultant object for q.

$$\overline{resp\_time} = \frac{1}{S}\sum_{i=1}^{S} \max_{j \in selec(i)} [T_{q_i,j} - T_{q_i}] \qquad (5)$$

Thus, the minimization problem is transformed into the following scheduling problem:

Problem 2 *Given a set of jobs p_1, p_2, $\cdots$, p_T, which take $selec(p_1)$, $selec(p_2)$, $\cdots$, $selec(p_T)$ time to finish, respectively. A set of queries q_1, q_2, $\cdots$, q_S, each of which is composed of a subset of these jobs. The association relation of jobs and queries is represented by a table (cf. Figure 5, "1" in cell (i,j) denotes q_j comprises p_i). A query is completed if and only if all of its associated jobs is finished. Find the optimal job sequence to minimize the sum of all the query completion time.*

The problem is proved to be NP-hard [6]. However, we show the following proposition on the optimal solution to the problem:

Proposition 1 *Let q_1, q_2, $\cdots$, q_S denote the completion order of the queries in the optimal job sequence. Any job sequence belonging to such sequence class: {jobs in q_1}, {remaining jobs in q_2}, $\cdots$, {remaining jobs in q_S} is an optimal sequence to Problem 2.*

Proof: See [6].

Proposition 1 indicates that finding an optimal order of the jobs is equivalent to finding the optimal completion order of the queries. Although the latter problem remains to be NP-hard, we can use the metric in Policy 1 as a reference to derive a suboptimal policy for collecting mode. However, two differences are noteworthy between these two policies:
1. Since we are to order queries instead of jobs, it's necessary to redefine the metric in Policy 1 in terms of query q. We define $sat(q)$ as the summation of the $sat(p)$ over all the jobs that are associated with q and $selec(q)$ as the summation of $selec(p)$, which still denotes the number of result objects in p for all active queries. Thus the metric for query q is, $\frac{\sum_{p\in q} sat(p)}{\sum_{p\in q} selec(p)}$;
2. Since $selec(p,q_i)$ and $selec(q_i)$ are unknown in Problem 2, $sat(p,q_i)$ is redefined based on its original semantics: the proportion of the objects in job p for query q_i. Therefore, $sat(p,q_i)$=$\frac{1}{intersec(q_i)}$, where $intersec(q_i)$ denotes the number of jobs that q_i is composed of (cf. Figure 5). The definition of $sat(p)$ is unchanged, $sat(p)$=$\sum_{i=1}^{S} sat(p,q_i)$.

In the example of Figure 5, $sat(p_1,q_1)$=1, $sat(p_2,q_2)$=1, $sat(p_2,q_3)$=$sat(p_3,q_3)$=1/2, $sat(p_1,q_4)$=$sat(p_2,q_4)$=1/2. Therefore, $sat(p_1)$=1+1/2=1.5, $sat(p_2)$=1+1/2+1/2=2, $sat(p_3)$=1/2. According to the $selec$ of each job p_i in Figure 5, the metric value of each query q is: $metric(q_1)$=$\frac{1.5}{1}$=1.5, $metric(q_2)$=$\frac{2}{4}$=0.5, $metric(q_3)$=$\frac{2+0.5}{4+2}$=$\frac{5}{12}$, $metric(q_4)$=$\frac{1.5+2}{1+4}$=0.7.

Hence, the scheduling policy for collecting mode is:

Scheduling Policy 2 *Collecting Mode Scheduling Policy: First, order all queries q_i in descending order of their $\frac{\sum_{j=1}^{intersec(q_i)} sat(p_{i,j})}{\sum_{j=1}^{intersec(q_i)} selec(p_{i,j})}$ values* [4]. *Then obtain the job sequence as: $p_{1,1}$, $p_{1,2}$, $\cdots$, $p_{1,intersec(q_1)}$, $p_{2,1}$, $\cdots$, $p_{S,intersec(q_S)}$.* [5]

In our example, queries are ordered as q_1, q_4, q_2, q_3. Therefore, the job sequence should be p_1, p_2, p_3.

5.3 Dynamic Scheduling Algorithms

In the previous two subsections, we analyze the scheduling policies for immediate and collecting modes in the static setting, i.e., scheduling the current queries in the active query table. However, as queries are dynamically inserted into and removed from the active query table, a runtime scheduling algorithm should also address the following issues:

- When do the metrics for the queries and jobs change so that the scheduler has to re-order the job sequence?
- For queries having deadlines (either soft or firm), how should the scheduler take these factors into consideration?

5.3.1 Scheduling Algorithm in Immediate Mode

In immediate mode, according to Policy 1, the values of $sat(p)$ and $selec(p)$ change if and only if a query is completed or a new query is inserted. Therefore, the scheduler needs to re-order all the jobs then.

For queries with deadlines, the immediate mode returns results continuously, a *soft deadline* set by the client is more reasonable. That is to say, an overdue query is still returned, although it's counted as a missed query. We tackle soft deadline through the notion of *query weight*. In Section 5.1, all queries are assigned equal (unit) weights. The mean response time is the arithmetic average of all query response times. If we dynamically assign weights to these

[4] $p_{i,j}$ denotes the jth job in query q_i.
[5] This sequence will remove any duplicate jobs because a job should only be scheduled once.

Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03)
0190-3918/03 $ 17.00 © 2003 IEEE

queries according to their closeness to deadlines and minimize the weighted mean response time, urgent queries will have higher priority in determining the order of jobs sequence. Thus, the soft deadline issue is handled without modifying the policy.

$$weight(q) = \begin{cases} \lceil (\frac{alerting\_period}{deadline(q)-t})^{\alpha} \rceil & t < deadline(q) \\ 1 & t \geq deadline(q) \end{cases} \quad (6)$$

Equation 6 describes the weight assignment function, where t denotes current time and $alerting\_period$ denotes the length of the time period ahead of the deadline when the weight of the query starts to increase (prior to this period, all query weights are 1 according to this equation). α is the weight increment ratio. In other words, it specifies the "firmness" of the deadline posed on the scheduler. The higher the value, the stronger the system requires deadlines to be met. Weight update is executed right before a job re-ordering is to be carried out to provide the most up-to-date query weights. The pseudocode of the immediate mode scheduler is described in Algorithm 1.

Algorithm 1 Runtime Scheduler for Immediate Mode

Input: $queue$: queue of jobs to be executed
$table$: the active query table
Procedure:
1: **while** SYSTEM_UP **do**
2: **if** queue is not empty **then**
3: job = queue.popFirst(); job.execute();
4: table.removeFinishedQueries();
5: **while** table.size $<$ S AND exist(nextQuery) **do**
6: table.insert(nextQuery); queue.insertJobs(nextQuery);
7: **if** table is changed OR system supports deadline **then**
8: update $weight(q)$ according to Equation 6
9: update $sat(p)/selec(p)$ for all jobs in $queue$
10: re-order all jobs according to Policy 1

5.3.2 Scheduling Algorithm in Collecting Mode

In collecting mode, from Proposition 1 and Policy 2, all jobs in the same query will be executed as a batch. Further, the metric $\frac{\sum_{j=1}^{intersec(q_i)} sat(p_{i,j})}{\sum_{j=1}^{intersec(q_i)} selec(p_{i,j})}$ will change if and only if a query is completed or a new query is inserted. This indicates that to lower computation overhead, each time the scheduler only needs to pick up the current best active query and execute all of its associated jobs in a batch.

Regarding deadlines, since in collecting mode, results are not sent back until all objects are retrieved, a hard (firm) deadline from the client is more reasonable. This means that an overdue query should not be returned and it's counted as a missed query. Therefore, when the scheduler chooses the best query to be executed, it first removes those queries that are already overdue or definitely going to be overdue [6]. It then chooses among those queries that are going to miss the deadline if not being executed right now [7]. These queries are called *urgent queries* in the sequel. In case more than one such query exist, the most urgent one (ordered by $deadline\text{-}execution\_time$ metric), will be chosen as the next query to execute. If no such query exist, the scheduler picks up the query with the highest $\frac{\sum_{j=1}^{intersec(q_i)} sat(p_{i,j})}{\sum_{j=1}^{intersec(q_i)} selec(p_{i,j})}$ value according to Section 5.2. The pseudocode of the collecting mode scheduler is described in Algorithm 2.

Algorithm 2 Runtime Scheduler for Collecting Mode

Input: $queue$: queue of jobs to be executed
$table$: the active query table
Procedure:
1: **while** SYSTEM_UP **do**
2: **while** queue is not empty **do**
3: job = queue.popFirst(); job.execute();
4: **while** table.size $<$ S AND exist(nextQuery) **do**
5: table.insert(nextQuery); queue.insertJobs(nextQuery);
6: **if** system supports deadline **then**
7: query.removeMissedQueries();
8: **if** $table$ has urgent queries **then**
9: nextQuery = table.MostUrgentQueries();
10: queue.insert(nextQuery.jobs);
11: **if** $queue$ is empty **then**
12: update $\frac{\sum_{j=1}^{intersec(q_i)} sat(p_{i,j})}{\sum_{j=1}^{intersec(q_i)} selec(p_{i,j})}$ for all queries
13: nextQuery = table.findBestQueries();
14: queue.insert(nextQuery.jobs);

6 Performance Evaluation

In this section, we compare our proposed query decomposing and scheduling approach (denoted as SCH) against sequential query execution[8] (denoted as SEQ) under various spatial datasets, workloads, query distributions, etc.

6.1 Simulation Testbed

We implemented a simulation testbed based on the model in Section 3. A population of mobile clients are moving in the universe. Their moving behavior is modeled according to *GSTD* [14], a well adopted spatiotemporal dataset generator. They issue window queries periodically and submit them to the attached MSSs, which in turn submit them to SDB. The mobile clients obey a *Zipf* distribution [11, 3]: they are inclined to locate in a set of "crowded" MSS cells. The arrival of queries at SDB is modeled as a *Poisson process*. A query window is a square with edge length r_{query} uniformly distributed between r_{max} and r_{min}.[9] For queries with deadlines, the time threshold is modeled as a *Gaussian* distribution.

[6] That is, $t + execution\_time > deadline$, here $execution\_time$ is calculated by the selectivity estimation.

[7] $deadline - average\_execution\_time < t + execution\_time \leq deadline$

[8] That is, it executes queries in a first-come-first-serve fashion. The detailed processing procedure for a window query is described in Section 2.2

[9] We assume the entire universe is a square unit.

Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03)
0190-3918/03 $ 17.00 © 2003 IEEE

Notation	Definition	Value
ppl	number of mobile clients	10000
$cells$	number of MSSs	1024
θ	client's *Zipf* skewness	0.95
r_{query}	query window's edge length	$r_{max} = 0.1, r_{min} = 0.05$
$t_{deadline}$	time threshold for query	$\mu = 2000, \sigma = 500$
$queries$	# of queries for each run	10000
S	active query table size	100 unless otherwise stated
λ	mean query arrival rate	in the range of $[0.0001, 0.005]$

Table 1. Parameters for the Simulation

We adopt two spatial datasets in our experiment, one synthetic dataset and one real dataset. The synthetic data (denoted as UN) obeys a uniform distribution with 100,000 data objects, while the real dataset [13] (denoted as CA) contains point locations of 62,556 California places. Table 1 summarizes the parameter settings in the simulation.

6.2 Performance Metrics

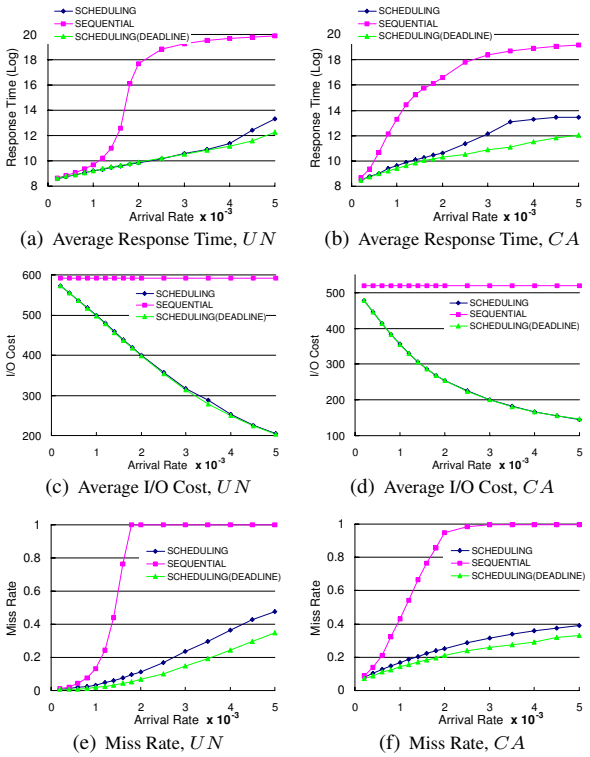
As our research focuses on I/O minimization, the following metrics are of interest to the performance evaluation:
1. **system throughput**: query processing rate, i.e., the number of completed queries in a unit of time;
2. **I/O cost**: mean I/O page operations per query, which comprises both index I/O and object I/O;
3. **mean execution time**: average service time for a query;
4. **mean response time**: see Equation 1 and 5 for *Immediate Mode* and *Collecting Mode* respectively;
5. **miss rate**: proportion of the missed queries, for systems supporting deadlines only.

Since I/O time is generally several orders of magnitude longer than CPU time, we only measure the former in our experiment, i.e., a disk page access costs 1 unit of time. Without loss of generality, we set each data object to the same size (i.e., 1 page). Therefore, "I/O cost" and "mean execution time" are essentially the same and "system throughput" can be derived directly from them. In the sequel, we use these three terms interchangeably.

6.3 System Performance in Immediate Mode

We compare the performance of SEQ and SCH in terms of the aforementioned metrics. We vary the mean arrival rate of the queries (λ) and query window size (r_{query}) to simulate different workloads for SDB.

Figure 6 illustrates the response time (in logarithmic scale), I/O cost and miss rate with respect to query arrival rates. From Figure 6(a) and 6(b), SEQ starts to deteriorate when λ exceeds its mean query service time μ. Nevertheless, SCH has a steady response time for a wide range of workloads since queries in SCH do not simply pile up one after one. Instead, they are parallelized, decomposed, and re-scheduled to minimize I/O cost. Thus more queries does not necessarily mean a longer execution time: more common objects are shared and thus more redundant I/Os can be eliminated. Figure 6(c) and 6(d) verifies this argument:



(a) Average Response Time, UN (b) Average Response Time, CA
(c) Average I/O Cost, UN (d) Average I/O Cost, CA
(e) Miss Rate, UN (f) Miss Rate, CA

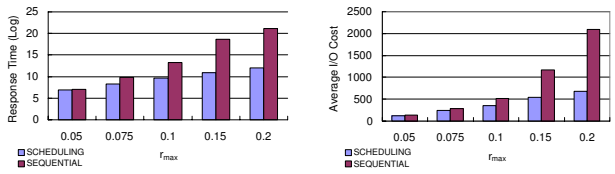
Figure 6. Performance v.s. Query Arrival Rates

as the query arrival rate is getting higher, the average I/O cost in SCH drops monotonously. We foresee that if λ approaches infinity and the active query table can hold all the queries, SCH will render an optimal, static job schedule in terms of I/O cost. On the other hand, if λ is much lower than μ, SCH degenerates to SEQ (cf. $\lambda = 0.0002$ in Figure 6(c) and 6(d)) because the number of queries in the active query table is so small that few objects can be shared as common results. The performance gain of SCH over SEQ in CA is less remarkable than that of UN. This is expected because our cost models assume that data objects are uniformly distributed inside a leaf MBR of the R-tree, which is the case for UN but not for CA.

For queries with deadlines, the weighted query algorithm in Section 5.3.1 (denoted by $SCH(DEADLINE)$) outperforms SCH by 15%-30% in terms of miss rate (cf. Figure 6(e) and 6(f)). Further, the response time and I/O cost of $SCH(DEADLINE)$ are still as good as SCH (cf. Figure 6(a), 6(c), 6(b) and 6(d)).

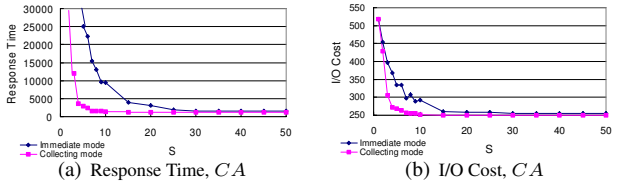
In the next experiment, we evaluate the response time and I/O cost in terms of various query window sizes. We vary r_{max} but keep r_{min}=$r_{max}/2$. Figure 7 shows that SCH always outperforms SEQ in different settings, and the larger the r_{max} value, the more performance gain is achieved: while the window size is increased by 16 times, the I/O cost of r_{max}=0.2 is only 5 times that of r_{max}=0.05 (cf. Figure 7(b)), because larger window queries have many more common data objects to share.

Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03)
0190-3918/03 $ 17.00 © 2003 IEEE



(a) Response Time, CA (b) I/O Cost, CA

Figure 7. Performance v.s. Query Window Sizes



(a) Response Time, CA (b) I/O Cost, CA

Figure 8. Impact on Active Query Table Size

6.4 System Performance in Collecting Mode

We conduct the same experiment for the collecting mode. The performance results are similar, thus not depicted due to space limitation. The major difference is that when firm deadlines are applied, i.e., overdue queries can be dropped, even SEQ has a much lower miss rate and response time for high arrival rates. In addition, the I/O cost of $SCH(DEADLINE)$ is a little worse than that of SCH since in $SCH(DEADLINE)$, the first priority is to prevent queries from being missed. Thus it sacrifices response time and I/O cost to miss rate.

6.5 Impact of Active Query Table Size

The active query table size S limits the number of queries that can be processed in parallel. The higher S is, the more redundant I/O cost is saved. However, S is limited by hardware resources: a large S requires both high computation (for selectivity estimation and job selection) and memory storage. Therefore, a moderate S is desirable to conserve these resources while keeping good performance.

We fixed λ to 0.002 for the real dataset. The results are shown in Figure 8. It's noteworthy that, both metrics converge at early stage in our settings: around S=10 for collecting mode and around S=20 for immediate mode. These small S values obviously will incur only a small overhead on memory and CPU resources, which confirms the practicability of our proposed system. In practice, the average memory storage for the active query table and in-memory R-tree is about 3MB. Regarding the computational cost, the average CPU time for scheduling is only 2ms for one query on a Pentium 4 desktop PC.

7 Conclusion

In this paper, we investigate how multiple window queries can be paralleled, decomposed, scheduled and processed in a realtime workload to optimize system performance for spatial database systems. We parallelize and decompose queries so that redundant I/Os to retrieve the same objects for different queries can be minimized. We also develop cost models and design dynamic schedulers to optimize system performance in terms of response time or miss rate for different result returning modes. Both deadline and no-deadline requirement are be supported. Empirical results show that our approach achieves a significant improvement over sequential query processing in various workloads. Meanwhile, experiments also show that the CPU and memory storage overhead is relatively small compared with the performance gain it obtains. Although the system is based on R-tree index, our query decomposition and job scheduling approach is not limited to R-tree — it can be applied to any partition-based spatial index, such as Quad-tree [2].

As part of our future work, we plan to extend our work beyond window queries to include other types of spatial queries, such as nearest neighbor queries. A more elaborated selectivity estimation method will be developed in order to render more accurate metrics for job scheduling.

References

[1] W. G. Aref and H. Samet. Optimization for spatial query processing. In *17th VLDB*, pages 81–90, 1991.

[2] M. Berg, M. Kreveld, M.Overmars, and O.Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 1997.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM (1)*, pages 126–134, 1999.

[4] M. N. Garofalakis and Y. E. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In *VLDB*, pages 296–305, 1997.

[5] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[6] H. Hu and M. Zhu. Towards real-time query processing in spatial databases. Technical report, HKUST, March 2003.

[7] H. Pang and M. J. Carey. Multiclass query scheduling in real-time database systems. *TKDE*, 7(4):533–551, 1995.

[8] H. Pang, M. J. Carey, and M. Livny. Managing memory for real-time queries. In *ACM SIGMOD Conference*, 1994.

[9] A. Papadopoulos and Y. Manolopoulos. Parallel processing of nearest neighbor queries in declustered spatial data. In *GIS*, pages 35–43, 1996.

[10] W. Smith. Various optimizers for single-state production. *Naval Research Logistics Quarterly*, pages 59–66, 1956.

[11] C.-J. Su and L. Tassiulas. Broadcast scheduling for information distribution. In *INFOCOM (1)*, pages 109–117, 1997.

[12] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *ACM SIGMOD Conf.*, 2002.

[13] Y. Theodoridis. Spatial datasets: an unofficial collection. http://dke.cti.gr /People/ytheod/research/datasets/ spatial.html, 2002.

[14] Y. Theodoridis and M. A. Nascimento. Generating spatiotemporal datasets on the www. *ACM SIGMOD Record*, 29(3):39–43, 2000.

Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03)
0190-3918/03 $ 17.00 © 2003 IEEE