Parallel Main-Memory Indexing for Moving-Object Query and Update Workloads

Darius Šidlauskas
Aalborg University
darius@cs.aau.dk

Simonas Šaltenis
Aalborg University
simas@cs.aau.dk

Christian S. Jensen
Aarhus University
csj@cs.au.dk

ABSTRACT

We are witnessing a proliferation of Internet-worked, geo-positioned mobile devices such as smartphones and personal navigation devices. Likewise, location-related services that target the users of such devices are proliferating. Consequently, server-side infrastructures are needed that are capable of supporting the location-related query and update workloads generated by very large populations of such moving objects.

This paper presents a main-memory indexing technique that aims to support such workloads. The technique, called PGrid, uses a grid structure that is capable of exploiting the parallelism offered by modern processors. Unlike earlier proposals that maintain separate structures for updates and queries, PGrid allows both long-running queries and rapid updates to operate on a single data structure and thus offers up-to-date query results. Because PGrid does not rely on creating snapshots, it avoids the stop-the-world problem that occurs when workload processing is interrupted to perform such snapshotting. Its concurrency control mechanism relies instead on hardware-assisted atomic updates as well as object-level copying, and it treats updates as non-divisible operations rather than as combinations of deletions and insertions; thus, the query semantics guarantee that no objects are missed in query results.

Empirical studies demonstrate that PGrid scales near-linearly with the number of hardware threads on four modern multi-core processors. Since both updates and queries are processed on the same current data-store state, PGrid outperforms snapshot-based techniques in terms of both query freshness and CPU cycle-wise efficiency.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*concurrency, parallel databases*

General Terms

Algorithms, Performance

Keywords

parallelism, spatio-temporal indexing

1. INTRODUCTION

In step with skyrocketing sales of smartphones and other mobile devices, the Internet is undergoing a rapid transformation into a predominantly mobile Internet. A range of technologies, e.g., GPS and technologies that exploit the communication infrastructure, enable the geo-positioning of mobile devices. As a result of this development, mobile location-based services stand out as a particularly successful class of Internet applications. This paper aims to contribute server-side infrastructure capable of efficiently supporting the kind of massive, location-related query and update workloads that result from the use of location-based services by large populations of Internet users, which we refer to as moving objects.

For instance, consider a country-wide traffic monitoring service. Assuming a population of 10M moving objects that move on average at 10 m/s and assuming that object positions need to be known with an accuracy of at least 100 m, this scenario entails up to 1M updates per second. If the population size or required accuracy are increased, so is the server-side update load. In addition to updates, the server must contend with queries that exploit the object locations in order to deliver a variety of services. As updates and queries can be expected to be correlated, so that frequently queried objects are also updated frequently, the workloads are challenging.

Workloads such as this cannot be sustained by disk-based techniques. Instead, we propose a main memory index that is capable of exploiting the inherent parallelism available in modern multi-core processors. The main challenge is to avoid the contention between queries and updates that conventional indexing and locking techniques use in order to maintain a consistent database state and return correct query results.

Two recent spatial indexing techniques remove conflicts between concurrent queries and updates. Both do this by maintaining two separate index structures—one for queries and one for updates. The techniques differ in how the structures for queries are kept reasonably up to date. TwinGrid [28] periodically replaces its query index with a copy of its update index. MOVIES [7] instead repeatedly accumulates updates and rebuilds its query index based on the updates accumulated since the last rebuild. Although TwinGrid outperforms MOVIES significantly [28], both techniques suffer from the problems of stale query results and wasted CPU cycles on full snapshotting.

Consider the application scenario described above. If, on average, 99% of a returned query result is required to be up to date, i.e., takes into account all the updates received before the query, index copying in TwinGrid must be performed every 100K updates. Empirical studies covered in this paper show that this results in approximately 50% of all CPU cycles being spent on copying. As 99% of the copied index entries do not change in-between any two copyings, this represents a substantial and unattractive waste of computing resources.

To address this deficiency, we propose a main-memory spatial index structure that avoids the trade-off between query freshness and CPU cost. With this new approach, queries return fresh results. This is achieved by the careful use of techniques that enable light-weight locking (e.g., 1-byte latches with the atomic CAS instruction), thus locking as little data as possible for as short time as possible, and avoiding the overhead of heavy-weight locks (e.g., 40-byte `pthread` mutexes).

The proposed indexing technique, called PGrid (parallel grid) is grid-based and processes queries and updates on the same data structure. Multiple copies of data are maintained at object granularity. Specifically, each object can have up to two versions of its position: its previous and current positions. Previous positions are kept only for objects that can otherwise be missed by queries due to object movement in the index structure. The technique exploits the spatial locality inherent to position updates to garbage collect previous positions of moving objects when they are no longer needed. In summary the contribution of the paper is three-fold:

1. We carefully consider query semantics for the parallel processing of moving-object workloads and propose a definition of query semantics that provides fresh query results as well as enables a high-degree of parallelism between queries and updates.
2. We propose a parallel main-memory indexing technique that offers three main advantages: (i) it provides up-to-date query results, as updates and queries operate on the same data; (ii) it wastes no CPU resources on frequent copying; likewise, the "stop-the-world" problem (interruption of workload processing) is avoided, and no CPU cache thrashing occurs due to snapshot building; (iii) the technique's concurrency control mechanism treats updates as atomic operations rather than as combinations of deletions and insertions, guaranteeing that no objects are missed by queries.
3. We report the results of extensive performance experiments on four different processors, demonstrating that our proposal has the best overall workload performance among existing moving-object indexing techniques.

Section 2 describes the problem setting and covers related work. Section 3 describes formally the query semantics supported by the proposed indexing technique. The index structure and algorithms are described in Section 4. Analytical and empirical studies of the proposal are covered in Section 5. Section 6 concludes the paper.

2. PRELIMINARIES

2.1 Problem Setting

We consider a setting in which a population of moving objects, be it mobile phone users or vehicles, are capable of reporting their positions to a central server that in turn delivers a variety of location-based services to the moving objects. We model the objects as point objects and the space in which they move as a two-dimensional Euclidean space. To support workloads consisting of queries as well as updates, the server employs a spatial index structure. While we focus on range queries, other types of queries, e.g., nearest neighbor queries, can easily be derived [12].

An update message includes the object's id (oid) and its new two-dimensional coordinates (x, y). Although the processing of an update from a single object can be performed efficiently, the tracking of a large population of objects with high accuracy subjects the server to extreme update loads. A rectangular range query is defined by its lower-left and upper-right corners, $(x_{q_{min}}, y_{q_{min}})$ and $(x_{q_{max}}, y_{q_{max}})$. Queries examine many objects and take much longer to process than do updates.

We assume updates occur according to shared-prediction-based tracking [6], which ensures that an object's position as known by the server is no further away from its real (measured) position than a given distance threshold, δ. At any time, any object is thus guaranteed to be in the circle with radius δ around its reported position. To capture all objects that might be in a query range, the range must be extended by δ[1]. If instead we want to capture all objects that are guaranteed to be in a query range, the range must be shrunk by δ. We assume that all incoming queries are already extended, shrunk, or left unmodified according to user requirements.

Consistent with other studies [7, 12, 21, 27, 30], we also assume that we know (possibly conservatively) the maximum possible speed of any object, v_{max}, e.g., 50 m/s [12, 21] or 60 m/s [7]. Together with the δ threshold, the minimum time between consecutive updates of an object then becomes $T_o = \delta / v_{max}$.

We aim to exploit the parallelism offered by modern multi-core and multi-threaded processors for the processing of the mentioned workloads. Unlike in single-threaded processing, queries and updates cannot be considered as instantaneous; rather, their processing occurs during some time interval $[t_s, t_e]$, and the intervals of operations can overlap.

2.2 Related Work

Much work has been done on the indexing of moving objects. We focus on the recent memory-resident indexes that support parallel processing. An extensive survey of spatio-temporal access methods proposed over the last decade can be found elsewhere [20]. We also focus on the indexing of current object positions.

Existing concurrency control (CC) protocols for multi-dimensional indexes [4, 14, 15, 19, 23, 25] consider only three index operations: search, insert, and delete. In our setting, it is necessary to also take into account an *update* operation that consists of the combination of a deletion and an insertion. This represents an additional complication for CC protocols. To increase concurrency, CC protocols often relax their search semantics. An example is to not consider phantom problems as critical [10, 25] and then to allow concurrent insertions and deletions in the search region. Several phantom-protecting CC schemes for multi-dimensional indexes have been proposed [4, 15]. However, they target disk-based indexes. Before multi-core CPUs, the key reason to support concurrent operations was to let some threads progress while others were blocked on I/O operations. With memory-resident data structures and multi-core CPUs, such opportunities no longer exist, and any complexity in CC transforms into overhead. To our knowledge, no efficient means of handling phantom problems exist for the setting we consider.

Hwang et al. [10] compare six main-memory R-tree variants, including their concurrent performance. They find that conventional latch-based CC using the link technique [15] (originally proposed for the B-tree [17]) does not scale: an 8-CPU machine is exhausted with just 6 threads under a simple update-only workload. Other benchmarks also suggest that lock-based approaches are inefficient [5]. Further, a study [10] shows that lock-based R-tree variants are outperformed by counterparts that utilize optimistic latch-free index traversal (OLFIT) [3]. Since we make use of the OLFIT technique, we describe it in detail later. Since R-trees inherently suffer from CPU-intensive update algorithms (splitting/merg-

[1]The Minkowski sum of the query range and a circle with radius δ must be performed.

ing of nodes, propagating bounding rectangle changes, etc.), we choose to use a grid-based index. This general approach has proved to be superior for update-intensive workloads in single-threaded environments [29].

One way to avoid interference between parallel queries and updates is to let queries and updates operate on different data structures [7, 28]. The query structure is read-only and so does not require locking. Single-object updates can be parallelized using atomic operations[2], by partitioning the data into disjoint sets and allowing only one update thread per set (shared-nothing [26]) or by using a simple concurrency control scheme [28]. This approach is capable of high performance and scalability on multi-core architectures, but as descried earlier, it has two drawbacks: (i) queries are based on stale data in the query data structure, and (ii) frequent copying of unchanged values when rebuilding the query data structure results in a substantial waste of CPU cycles.

Based on a *copy-on-write* mechanism, a granular rebuilding technique has been proposed [23]. When updates are about to alter parts of the (query) data structure (the T-tree), new copies of those parts are created and modified off-line. This enables both updates and queries to traverse the data structure without latches. The integration of fresh parts into the structure is relatively cheap as only pointers have to be swapped. However, the high cost of creating versions (memory allocation/deallocation, data copying) renders this technique useful only for modest update rates.

3. SEMANTICS AND PARALLELISM

In a single-threaded execution, at any point in time either a single query or a single update is being processed. In a multi-threaded setting, system throughput can be increased by means of *intra-operation* and *inter-operation* parallelisms. Since we are faced with very simple single-object update operations, our focus is on inter-operation parallelism, where different operations are processed in parallel. In the following, unless noted otherwise, an update is a transaction consisting of a deletion of an object's old position and an insertion of its new position. When objects enter or leave the system, an update can also be a single insertion or a single deletion.

Parallel execution of queries and updates naturally raises the question of query semantics, i.e., how the correct result of a query is defined. As in traditional databases, serializability of a parallel execution of operations is a desirable property, but query semantics corresponding to a lower level of isolation between transactions enables more parallelism. In the following, we discuss two different semantics and show that slightly relaxing the semantics increases the potential parallelism significantly and is still perfectly acceptable for the applications we consider. The findings in this section are general in nature and are not restricted to a particular indexing technique.

3.1 Full Serializability (Timeslice Semantics)

A concurrency control scheme that ensures a complete isolation between update and query operations can be implemented as follows. An update, operating on a single object, obtains an exclusive lock on that object. The lock is released as soon as the update completes. A query obtains shared lock(s) on the range(s) of the data space that must be accessed to produce its answer [8]. Such a range usually includes multiple objects, and the lock on a range is released as soon as it has been accessed by the query. When queries

[2]E.g., at the expense of accuracy, object data can be packed into 64-bit values. Then object reads and writes are atomic on a 64-bit architecture.

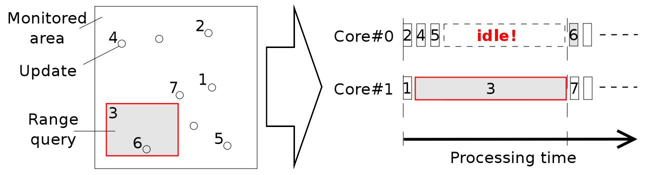


Figure 1: A long-running query holds back rapid updates. Numbers indicate arrival order.

and updates operate concurrently on the same data, the queries must wait for the updates holding exclusive locks, which prevents them from reading inconsistent states. Similarly, the updates must wait for the queries holding shared locks. This limits the potential parallelism significantly. As shown in Figure 1, despite available resources on multi-core CPUs, rapid updates can be delayed by the time it takes to process a prior long-running query.

When index structures are used, updates may involve index structure modifications. Therefore, locks in concurrent spatial indexes are usually acquired at a much coarser granularity than that of single objects. For instance, in tree-based indexes an entire sub-tree or individual nodes can be locked [15, 19], while in grid-based approaches, entire cells or buckets (data pages) can be locked [24]. This results in even worse update/query interference. Past studies [5, 10] also suggest that workloads with both queries and updates do not scale in such a setting: only few cores are utilized efficiently.

The above locking protocols enable full serializability and implement *timeslice* semantics, meaning that a query reports precisely the objects within its range at a specific time instance, usually just after the query processing starts (and the necessary locks are acquired). This is equivalent to degree 3 consistency in database systems [9].

To illustrate the performance penalty a naive implementation of timeslice semantics may cause, consider the traffic monitoring scenario. In a single-threaded setting, an update takes circa 1 microsecond, while a query might take up to few milliseconds depending on the size of the result [29]. This implies that an update might be delayed by three orders of magnitude longer than its actual processing time.

3.2 Freshness Semantics

Traditional CC schemes that have been shown to scale for main-memory R-trees [10] ensure only the consistency of the underlying index in the presence of three concurrent operations: searches, insertions, and deletions. In these schemes, insertions and deletions are allowed in a concurrently searched region, i.e., no range or predicate locking is used. Assume that for the setting we consider, to prevent partial reads and writes of object data, a CC scheme is modified to ensure the atomicity of updates (deletion-insertion pairs). In such a setting, which semantic guarantees are sacrificed, when compared to timeslice semantics?

Assuming that the query processing lasts from t_s to t_e, Figure 2 shows a snapshot at some point in time between t_1 and t_2 ($t_1, t_2 \in [t_s, t_e]$) of simultaneous range querying and updating. At t_1, the query has already inspected half of its range (the grey area), and processing is in progress (striped area). Black dots are object positions at the beginning of the query (at t_s), and white dots are their updated positions that were processed during $[t_1, t_2]$. We can identify four inconsistencies in the sketched, simplified CC scheme when compared to timeslice semantics:

(i1) Object A is within the query range at t_s. However, it exits

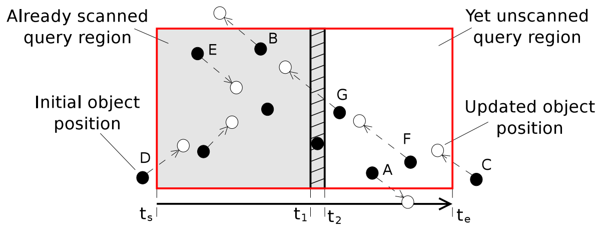


Figure 2: Parallel updating and querying.

the range before being seen by the query and therefore is not reported. With timeslice semantics, A is a false negative. Note that B also exits, but is captured in both CC schemes.

(i2) Object C is not within the query range at t_s. However, it is reported as it enters the range during (t_1, t_2). With timeslice semantics, C is a false positive. Note that D also enters the range during $[t_1, t_2]$, but is not reported in both CC schemes.

(i3) Some of the reported object positions are fresher than others. For example, consider objects E and F. They both are in the query range before and after their updates. However, the query reports only F's updated position.

(i4) Both of object G's positions are in the range, but the query fails to capture G because the update moves it from the as yet unscanned to the already scanned query region.

We argue that cases i1, i2, and i3 can be easily tolerated in the targeted application domains. In fact, the behavior in these cases may often be preferred in the context of continuously changing data because the behavior implies that freshness of results is preferred over returning results that were consistent (according to the timeslice semantics) as of the start of the query.

However, case i4 is unlikely to be acceptable for any application domain: the query misses objects although they are always in its range. This is due to not locking the query range. To remedy this problem, we assume that $t_e - t_s < T_o$, meaning that the time needed to process a query is shorter than the time between two consecutive updates of an object. This assumption is true for most realistic settings in our application domain. In the traffic monitoring example, given a maximum object speed v_{max} of 216 km/h and a required high accuracy δ of 10 m then $T_o = 170$ milliseconds. This enables the processing of very long-running queries, given the main memory setting. In pathological cases, where a query does not meet this requirement, a simple timer can be maintained. Then, if a query does not complete within a given time, it can be restarted or aborted. We shall later see that typical queries complete in times that are a few orders of magnitude shorter than 170 ms.

With the above assumption, any examined object can be updated *at most once* during the time $[t_s, t_e]$ when a query is processed. Therefore, case i4 can be handled by keeping one previous object position in the index. As such, we are guaranteed that a query always encounters at least one object version and so does not miss any objects. In Figure 2, the query will report G's previous position (black dot).

We call the level of guarantees that corresponds allowing cases i1 to i3, but disapproving case i4, *freshness* semantics: a query, processed from t_s to t_e, returns all objects that have their last reported positions before t_s in the query range, and it reports *some* (fresher) objects that have their last reported positions after t_s (and before t_e) in the query range. Below, we define the freshness semantics formally. We use $pos^\bullet$ and pos° to denote the previous and current position of an object, respectively, and t_u denotes the last time an object was updated.

Definition 1. Given a query $\Re$ with processing time $[t_s, t_e]$, its result O is said to satisfy freshness semantics if for any object o, the following holds:

1) if $o.t_u < t_s$ then $o \in O$ if and only if $o.pos^\circ \in \Re$
2) if $t_s < o.t_u < t_e$ then
   a) if $o.pos^\bullet \in \Re$ and $o.pos^\circ \in \Re$ then $o \in O$
   b) if $o.pos^\bullet \notin \Re$ and $o.pos^\circ \notin \Re$ then $o \notin O$
   c) if $o.pos^\bullet \notin \Re$ and $o.pos^\circ \in \Re$ then o may or may not belong to O
   d) if $o.pos^\bullet \in \Re$ and $o.pos^\circ \notin \Re$ then o may or may not belong to O

The first part says that if o was only updated before the query started then whether o is in the query result is determined by its up-to-date position. The second part deals with objects that are updated during query processing and covers the cases discussed already. Observe that conditions (2.c) and (2.d) imply that if one position is within the query range while the other is not, the decision to add o to the result is arbitrary. This is because not necessarily all updates after t_s are seen by the query during $[t_s, t_e]$. For example, the query might observe only o with $pos^\bullet$; then $o \in O$ if $o.pos^\bullet \in \Re$ (while its actual pos° might be inside or outside the query range).

While parallel executions that satisfy freshness semantics are not guaranteed to be serializable[3], it can be easily seen that, if only one query is considered with a mix of updates, freshness semantics implies serializability.

The PGrid indexing technique, detailed in Section 4, is capable of exploiting freshness semantics.

4. PARALLEL GRID

Here, we detail the *PGrid* indexing technique that implements freshness semantics. First, we outline the index structure, the operations supported, and the types of locks used. Next, we cover the update and query algorithms, followed by specifics on how to implement atomic reads and writes of single-object data.

4.1 Structure

PGrid exploits an existing main-memory index structure that offers high performance for traffic monitoring applications in single-threaded settings [29]. Queries are serviced using a uniform grid [1], while updates are facilitated via a secondary index in a bottom-up fashion [16]. Figure 3 depicts PGrid's components and their structure.

A two-dimensional array represents a uniform grid directory that statically partitions a predefined region into cells. Objects with coordinates within the boundaries of a cell belong to that cell. The grid directory does not store any data. Rather, object data is stored in a linked lists of buckets, and a grid cell stores just a pointer to the first bucket of such a list, or the null pointer if no objects are in its cell. A bucket is fully described by three meta-data fields: a pointer to the next bucket (nxt), an integer specifying the number of currently stored objects (nO), and an integer specifying the number of readers currently scanning the bucket (nR). The former two fields are self-explanatory. The latter is used for reference counting and is explained later.

[3]The following execution satisfies freshness semantics, but is not serializable: When two queries Q_1 and Q_2 are executed in parallel, two updates A and B happen, both inside of the ranges of both queries. Query Q_1 takes into account only A, while Q_2 takes into account only B.

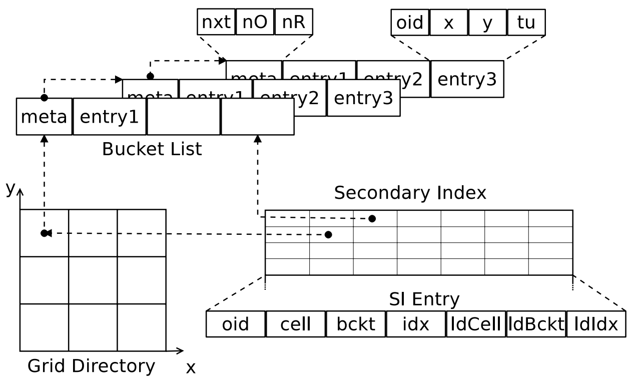


Figure 3: PGrid Structure.

The data for an object in the index is stored as a four-tuple with an object identifier, coordinates, and an update timestamp: (*oid, x, y, tu*). To comply with freshness semantics, at most two tuples can be associated with the same object: one representing the previous version and the other representing the most up-to-date version of the object.

Single-object updates are facilitated via a secondary index structure (e.g., a hash table) that indexes objects on their oid. Each secondary index entry provides information about an object in the primary structure. A cell pointer ($cell$) indicates the cell an object belongs to. A bucket pointer ($bckt$) together with a positional offset (idx) provides direct access to the actual object data in a bucket. This eliminates the need for cell or bucket scanning during updates. The same information is maintained for determining a tuple with the previous object location ($ldCell$, $ldBckt$, $ldIdx$), where the ld prefix indicates logical deletion, to be explained in the following.

As in the earlier single-threaded proposal [29], all incoming updates are categorized as *local* or *non-local*. When an object's new position belongs to the same cell as its currently stored position, the update is local; otherwise, the update is non-local and involves deletion of the object from its current cell and insertion into a new cell. Given that there is no need for the explicit object deletion and insertion operations during local updates, the updater simply overwrites the outdated object data (the coordinates and the timestamp). Since no structural modifications or data movements occur during a local update, a query cannot miss an updated object. It must only be ensured that single-object reads and single-object writes are atomic so that partial reads and writes of object data are prevented. In contrast, non-local updates require that previous object positions are kept, as such updates may move an object from the a cell yet to be scanned by a query to a cell that has already been scanned (see Figure 2).

To handle non-local updates, PGrid introduces a notion of *logical deletion*. A non-local update then does not actually remove an object's old position but, instead sets the update timestamp of the old position to the current time and marks the position with a deletion flag (implemented as the negation of the update timestamp). A logically deleted position is physically deleted with the next update of the object. Note that both logical and physical object deletion requires an update of the object's entry in the secondary index. A logical deletion initializes the fields $ldCell$, $ldBckt$, and $ldIdx$ to the object's current cell, bucket, and offset within the bucket, respectively, while a physical deletion nullifies them.

With parallel processing, the two data structures—the primary index (the grid) and the secondary index (the hash table)—are modified concurrently by multiple updaters. The changes made in one have to be reflected in another. To guarantee the consistency between the two, PGrid's concurrency control includes two types of locking: object locking and cell locking. The locks are independent from each other in the sense that cell locking does not block the cell's objects: the objects in a locked cell can still be accessed and modified individually. Both types of locks are stored separately from the PGrid structure.

The main purpose of object locks is to provide synchronized, single-object updates between the two structures. After an object lock is acquired, the updater is sure that the object-related data is not changed in either index by concurrent operations of other updaters. Since an object lock blocks access just to one particular object, it has a modest effect on the potential parallelism. As mentioned before, the server rarely, if ever, encounters concurrent updates to the same object.

The main purpose of a cell lock is to prevent concurrent cell modifications, i.e., physical deletion/insertion of new objects in a cell and, consequently, deletion/insertion of new buckets in the cell. For example, when a bucket becomes full, the cell lock guarantees that only one thread at a time allocates a new bucket and modifies the pointers so that it becomes the first. Cell locks are required only during physical object insertions and deletions. As detailed in the following, both operations are processed sequentially, implying that an updater can lock only one cell at a time. Therefore, no deadlocks are possible due to cell locking. Since a cell lock does not block other threads from accessing the objects in its cell, the objects involved in the deletion/insertion are locked individually using object locks.

We proceed to describe the update and query algorithms in detail. We show that freshness semantics can be supported without updates and queries being able to block each other.

4.2 Update Processing

Listing 1 shows the pseudo-code of PGrid's update algorithm. The algorithm takes as an input a new object tuple (new). Based on its position, a new cell for the object is computed. Then, an object lock is acquired (Line 2) and held until the end of the update (Line 15). After a secondary index look-up, in Line 3, the primary index related information is retrieved: the current cell of the object ($oldCell$) and the location of the object tuple (obj). No cell or bucket scanning is required.

Next, the updater checks whether the object has a tuple with a previous object location, i.e., whether the object was previously logically deleted. If this is the case, the fields $ldCell$, $ldBckt$, and $ldIdx$ in the secondary index are non-null and the tuple has to be (physically) deleted (Line 7). To be discussed later, to avoid deadlocks, a deletion might fail. In this case, the object lock is released and the update is restarted.

The update type is determined by comparing the new and old cells (Line 10). If the update is local, the outdated object tuple is overwritten with the new one (Line 11). Note that to ensure lock-free querying, the object write must be instantaneous. Section 4.4 describes how to do that.

If the update is non-local, the new tuple is inserted into the newly computed cell, and the outdated tuple is logically deleted (Lines 13–14). A logical deletion consists of two parts (not shown). First, the secondary index entry is updated: the fields referring to the current object tuple ($cell$, $bckt$, idx) are copied over the fields referring to the logically deleted tuple ($ldCell$, $ldBckt$, $ldIdx$). Second, the timestamp of the outdated tuple is set to the current time ($new.ts$). Also, to mark that it is a logically deleted version, the timestamp is negated. This informs the query threads that the tuple contains a previous object position.

Listing 2 contains the pseudo-code for physical object deletion.

Listing 1: `update`(ObjectTuple new)

```
1  newCell = computeCell(new.x, new.y);
2  lockObj(new.oid);
3  sie = SecondaryIndex.lookup(new.oid);
4  oldCell = sie.cell;
5  obj = getObj(sie.bckt, sie.idx);                  // object tuple
6  if hasLD(sie) then
7      if !delete(sie) then                          // physical deletion
8          unlockObj(new.oid);
9          go to 2 ;                                 // try again
10 if newCell == oldCell then
       /* Local update */
11     writeObj(new, obj);                           // new copied over obj
12 else
       /* Non-local update */
13     insert(new, newCell, sie);                    // physical insertion
14     LD(sie, new.tu);                              // logical deletion
15 unlockObj(new.oid);
```

Listing 2: bool `delete`(SIEntry sie)

```
1  lockCell(sie.ldCell);
2  ldObj = sie.ldBckt.entries[sie.ldIdx];
3  firstBckt = *sie.ldCell;                  // cell refers to the 1st bucket
4  lastObj = firstBckt.entries[firstBckt.nO - 1];
5  if tryLockObj(lastObj) then
6      writeObj(lastObj, ldObj);             // lastObj → ldObj.
7      firstBckt.nO- -;                      // decrement
8      if firstBckt.nO == 0 then             // is empty?
9          *sie.ldCell = firstBckt.nxt;
           // No more queries can enter firstBckt
10         waitUntilNoReaders();
11         free(firstBckt);
12     Nullify all ld references in sie ;
13     Lookup for lastObj's sie and update it;
14     unlockObj(lastObj); unlockCell(sie.ldCell);
15     return true;
16 else
17     unlockCell(sie.ldCell);
18     return false;
```

Listing 3: `insert`(ObjectTuple new, Cell cell, SIEntry sie)

```
1  lockCell(cell);
2  firstBckt = *cell ;                       // cell refers to the 1st bucket
3  if isFull(firstBckt) then
4      Allocate new bucket and make it first ;
5  freePos = firstBckt.entries[firstBckt.nO];
6  writeObj(new, freePos);                   // new → freePos
7  firstBckt.nO++;                           // increment
8  Update cell, bckt, and idx in sie ;
9  unlockCell(cell);
```

The algorithm takes the secondary index entry as input, and it reports whether or not the deletion was successful. In a nutshell, the deletion algorithm tries to move the last object of the first bucket into the place of the object to be deleted. The processing is secured by a cell lock. Again, the secondary index provides the necessary information on the logically deleted object without any scanning (Lines 1–4). After the last occupied entry of the first bucket is determined ($lastObj$) and is successfully locked (Line 5), the logically deleted object is overwritten (Line 6). Note that for a very short time, until the bucket's counter is decremented (Line 7), a query thread can see two identical tuples for $lastObj$. The query algorithm takes this into account (see Section 4.3).

If the first bucket becomes empty, it is removed, and the next bucket becomes the first, or the grid cell becomes empty. However, since it is possible for concurrent queries to be scanning the bucket, the deleter "busy" waits until all queries leave the bucket (Line 10) and only then deallocates its memory (Line 11).

Then, the secondary index is updated. All pointers to the logically deleted object are nullified (Line 12). The secondary index is also searched for the $lastObj$'s entry so that it can be updated to store its new position in the grid index (Line 13). Depending on whether the $lastObj$ tuple contains a logically deleted position or an up-to-date object position, the fields *bckt* and idx or *ldBckt* and *ldIdx* are modified accordingly. Eventually, the acquired locks are released and the deleter returns successfully.

A failure to lock $lastObj$ in Line 5 means that it is already locked by another thread. Instead of blocking and waiting until it can be obtained, the deleter unlocks the previously locked cell and returns with a failure indication (Lines 17–18). The unsuccessful return forces the update algorithm to restart its processing (Listing 1). This costly decision eliminates a potential deadlock. The deadlock would happen if both of the following two circumstances were to occur while running `delete`. First, another concurrent updater has to be updating the same $lastObj$ object, implying that it is already locked at Line 2 of Listing 1. Second, the $lastObj$ tuple must be a logically deleted object so that the concurrent updater also needs to enter the deletion routine and thus need to obtain a lock on the same cell. The two updaters end up waiting for each other. The restart of one the updaters solves the problem. Situations such as this are unlikely to occur, and our empirical study confirms that restarts are very rare.

Physical object insertion is relatively simple (Listing 3). A new object is always inserted at the end of the first bucket, which is pointed to by the cell (Line 2). In case it is full, a new bucket is allocated, and the necessary pointers are updated so that this bucket becomes the first (Line 4). The first free position at the end of the bucket is determined (Line 5), and the new tuple is written (Line 6). As the processing is secured by the target cell lock, other inserters cannot write to the same position.

4.3 Query Processing

PGrid naturally supports object-id queries using its secondary index on object oid. We proceed to describe range query processing that satisfies freshness semantics. Note that other types of queries can also be supported. For example, k-nearest neighbor queries can be derived from range queries [12].

The range query algorithm partitions the cells overlapping the query range into two sets: the fully and the partially covered cells. Only the objects in partially covered cells need to be checked to see whether they are in the range. Both types of cells are scanned without object locks, although each cell is locked briefly before entering its first bucket. An object's timestamp value is used to distinguish between the two copies of the object. Listing 4 provides the details, which we proceed to describe.

The algorithm takes three inputs: a two-dimensional rectangle specifying the query range, an integer value as the query's timestamp (t_s), and a container for storing the result. For simplicity, Listing 4 shows only the processing of fully covered cells. Thus

Listing 4: `rangeQuery`(Rect q, int ts, Container res)

```
1  cells = computeCoveredCells(q);
2  foreach cell ∈ cells do
3      lockCell(cell);
4      if isEmpty(cell) then
5          unlockCell(cell);
6          continue;
7      bckt = *cell;                              // cell refers to the 1st bucket
8      aInc(bckt.nR);                             // atomic increment
9      unlockCell(cell);
10     while bckt != null do
11         for idx = bckt.nO - 1 downto 0 do
12             obj = readObj(bckt.entries[idx]);
13             if obj.tu > 0 then
14                 res.add(obj);                  // replaces if exists
15             else if abs(obj.tu) > ts and
               !res.contains(obj.oid) then
16                 res.add(obj);
17         if bckt.nxt != null then
18             aInc(bckt.nxt→nR);                 // atomic increment
19         aDec(bckt.nR);                         // atomic decrement
20         bckt = bckt.nxt;
21 Similar processing continues for partially covered cells;
```

the extra checking done for partially covered cells does not appear in the listing.

To prevent concurrent updaters from deleting a bucket that is about to be scanned by a query thread, query threads increment the reader counter (nR) of a bucket before entering it and decrement it as soon as the bucket has been scanned. Since the first bucket can have been deleted by the time the query actually increments its counter (Line 8), a cell lock is acquired briefly. For the subsequent buckets, the counter can be accessed safely, as the next bucket cannot be freed before the first one (Lines 17–20). Atomic operations are used because the counters can be accessed by multiple queries in parallel[4].

Recall that an updater can move the last object of the first bucket to its beginning (to overwrite an object to be deleted). If a query scans a bucket from its beginning, the moved object could be missed because it is moved from the as yet unscanned part of the bucket to the already scanned part. To eliminate this problem, objects within a bucket are examined starting from the last entry in the bucket (Line 11).

PGrid uses the update timestamp to distinguish between multiple copies of the same object. A positive timestamp signals that the tuple contains the most up-to-date object location and, according to the freshness semantics, must be taken into account (Line 14). A container for storing the query result is implemented so that adding replaces the previously stored object with the same *oid*, if it exists[5].

A negative timestamp signals that the tuple contains the previous object location. Such a tuple is added to the result if two conditions hold. First, the absolute value of its timestamp exceeds that of the query's ($|t_u| > t_s$). This implies that the object was updated after the query started and so can be missed (see case i4 in Section 3.2). Second, the query has not yet seen the object's new (updated) position. The tuple is simply ignored if the result already contains an object with the same id.

The following theorem states the correctness of the presented query and update algorithms.

THEOREM. *The* `rangeQuery` *algorithm (Listing 4), performed in the presence of concurrent* `update` *operations (Listings 1–3), returns results that satisfy freshness semantics.*

For brevity, we omit the proof. Its most important part is to show that case i4 from Section 3.2 is avoided. This is demonstrated based on the observation that an object may be relocated in the index structure (and, thus, potentially may be missed by a concurrent query) only if (i) the object is updated non-locally or (ii) it is the last object of the first bucket and it participates in a deletion of another object. Case ii is handled due to the scanning direction in the query algorithm (Line 11). In case i, the query will see at least one of the two positions of the object, the current or the previous, because the previous position of the object will not be garbage collected during the execution of the query due to the assumption that an object can be updated only once during the execution of a query, i.e., $t_e - t_s < T_o$ (see Section 3.2).

4.4 Parallel Object Data Reads and Writes

So far, we have assumed that single-object reads (`readObj`) and writes (`writeObj`) are performed in an atomic manner. This section shows how this can be achieved.

4.4.1 Background

On 64-bit architectures, the reading and writing of 8-byte aligned quadword values are guaranteed to be atomic. However, moving object data involves more bits per object (id, coordinates, timestamp). One approach is to pack the data to fit into a 64-bit value, but this occurs at the expense of extra computation in updating [13] and accuracy [7] in retrieval. In PGrid, we need 128 bits per object so multiple read and write operations have to be considered. Since moving-object workloads involve millions of rapid single-object updates, explicitly locking and unlocking of individual objects might incur a significant overhead.

To quantify the overhead, we conducted a micro-benchmark on the Intel Nehalem processor, where the CPU cycles needed to read and write a 128-bit value were measured. Table 1 shows the results. The single-threaded columns show the performance in a zero contention case, whereas the multi-threaded columns correspond to the performance in a highly contended case, as two concurrent threads—a reader and a writer—are constantly accessing the same 128-bit value.

A simple multi-read/multi-write (two 64-bit reads/writes) consumes 6 CPU cycles and is correct only in the single-threaded case. For the multi-threaded case, we first profiled the performance of the lock-based approaches using three types of synchronization defined in the Pthreads standard [22]. The mutually exclusive lock, `pthread_mutex_t`, allows only one thread at a time to lock and access the data. Next, the read/write lock, `pthread_rwlock_t`, allows several reader-threads to acquire locks in shared mode, but allows only one writer-thread to acquire a lock in exclusive mode.

[4]The cell locking in the query algorithm is related to safe memory reclamation that can be implemented completely lock-free using advanced techniques such as atomic double compare-and-swap (DCAS) operations (that are, however, not supported by commodity hardware) or by multi-threaded memory allocators (which are implemented as complex libraries). Since we did not observe any noticeable contention or performance penalty due to this cell locking, we do not consider such techniques.

[5]For small queries, a simple array can be used that is scanned before adding a new item. For large queries, any unique associative container, e.g., a hash map, will work.

Table 1: CPU cycles per 128-bit read/write

	single-threaded		multi-threaded	
Method	read	write	read	write
multi-read/write	6	6	-	-
`pthread_mutex_t`	46	46	358	374
`pthread_rwlock_t`	91	86	716	772
`pthread_spinlock_t`	27	27	98	128
1-byte latching using CAS	25	25	108	144
OLFIT	9	64	251	247
SIMD	4	4	25	21

Finally, a spin lock, `pthread_spinlock_t`, is a mutually exclusive lock where a thread spins in a loop, repeatedly checking until the lock becomes available ("busy wait"). All three locks are initialized with default attributes.

As expected, lock-based reads/writes using the mutexes available in the `pthread` library are expensive (row 1 vs. rows 2–4 in Table 1). The multi-threaded case shows how performance degrades further due to cache coherency protocols [18]. Since the actual reads/writes are fast (6 cycles) and a lock is held for a very short time, spin locks perform the best in both the single- and the multi-threaded case.

Looking beyond running time performance, one mutex for each object results in a significant memory overhead. For instance, the Pthreads mutex implementation occupies 40 bytes, which is 2.5 times the actual moving object data size (16 bytes). Therefore, we implemented 1-byte latches using the atomic CAS instruction and used them instead of Pthread synchronization in our implementations (calls to lock/unlock objects and cells in Listings 1–4). As Table 1 shows, its performance is very similar to that of spin locks.

Nevertheless, such fine grained locking is very inefficient during query scanning: the lock calls themselves incur a very substantial overhead over the actual reads/writes and adversely affects parallel update processing. In the following, we present two lock-free methods that are used for parallel reads and writes of moving object data in PGrid.

4.4.2 OLFIT

Optimistic lock-free index traversal (OLFIT) was designed as a cache-conscious concurrency control scheme for main-memory indexes on shared-memory multiprocessor machines [3]. This approach maintains a latch and a version number for each object. Table 2 depicts the read and write algorithms.

The OLFIT approach guarantees consistent reads and writes as follows. An updater always obtains a latch first, so no multiple writes are allowed on the same data item. In addition, before a latch is released, an updater increments the version number. A reader starts by copying the version number and optimistically reads the data without latching. Then, if the latch is free and the current version number is equal to the copied one, the read is consistent; otherwise, the reader starts over again.

The optimistic reads are especially favorable in multi-core architectures because they avoid the memory write required by latching. Thus, even if the actual object data does not change during parallel query processing, the entire cache block containing the latch is invalidated. This implies that other cores, with the same cache line cached in their local memory, are subject to coherence cache misses.

In the single-threaded case, OLFIT reads show up to ten times better performance than the lock-based reads (see Table 1). However, OLFIT writes, due to extra operations (latch/unlatch using relatively expensive atomic CAS instruction plus version increment), are more expensive than writes using `pthread_mutex_t` and `pthread_spinlock_t`. In the multi-threaded case, the OLFIT reader is often required to re-read content because of the concurrent updater-thread. As a result, OLFIT reads perform similar to OLFIT writes. The spin-lock and 1-byte latching methods outperform the OLFIT approach in the multi-threaded case.

Table 2: Read and Write in OLFIT

Read	Write
R1: copy the value of *version*	W1: acquire *latch*
R2: read the content	W2: write the content
R3: if *latch* is locked, go to R1	W3: increment *version*
R4: if current *version* differs from the copied, go to R1	W4: release *latch*

In the applications we target, the server rarely, if ever, encounters two concurrent update messages operating on the same object. So no contention is expected in step W1. Also, since single-object reads and writes are fast (6 CPU cycles on average), we can expect the conditions in steps R3 and R4 to fail only rarely. Therefore, we expect the OLFIT performance to be close to the single-threaded scenario in practice.

4.4.3 SIMD

Current commodity processors support the so-called single instruction multiple data (SIMD) technology. This makes it possible to achieve data-level parallelism via vector operations on multiple data items. For example, with the 128-bit wide SIMD registers on the Intel Nehalem processor, one can add four pairs of 32-bit integers simultaneously [11].

The instruction sets of such processors come with instructions for loading/storing data into SIMD registers. Thus, we can employ the SIMD technology in PGrid for parallel object data reads and writes. Since SIMD reads and writes are atomic, no explicit latching or locking are required. Table 1 shows that SIMDized reads and writes are the most efficient, outperforming also the lock-free multi-read/write in the single-threaded case. In the single-threaded case, it takes 4 cycles for both read and write, while in the multi-threaded case, due to the cache coherency protocol, the cost increases to 25/21 cycles per read/write.

5. PERFORMANCE STUDY

To explore the performance of PGrid, we compare it experimentally with a number of alternatives. Before reporting the findings, we describe the experimental setting and explore, analytically and experimentally, how to set the snapshotting period for TwinGrid [28], so as to achieve a fair comparison.

5.1 Experimental Setup

We study the performance on diverse multi-core platforms: a dual quad-core AMD Opteron 2350 (Barcelona) with 8 hardware threads, an 8-core Sun Niagara 2 (T2) with 64 hardware threads, a dual quad-core Intel Xeon X5550 (Nehalem) with 16 hardware threads, and a quad-core Intel Core i7-2600 with 8 hardware threads (Sandy Bridge). Caches are shared by all threads in a core or an entire chip. All machines have enough main memory to store both the entire workload and the populated index.

Index implementations All indexing techniques were implemented in C/C++ and compiled with g++ under the maximum optimization level. Five grid-based index implementations are considered: a single-threaded grid-based index (u-Grid) [29], its state-of-the-art multi-threaded variant (TwinGrid) [28], two variants of PGrid (PGridSIMD and PGridOLFIT), and a multi-threaded variant of u-Grid with timeslice semantics, which we call Serial.

Serial is a simple implementation of the timeslice semantics de-

Table 3: Workload Configuration

Parameter	Values
objects, x10^6	5, **10**, 20, 40
updates, x10^6	100
monitored region, km^2	Germany, 641×864
# road network segments	32,750,494
# road network nodes	28,933,679
speed$_i$, km/h	20, 30, 40, 50, 60, 90
range query size, km^2	0.25, 1, **4**, 16
update/query ratio	250:1, ... **1000:1**, ... 16000:1
time between updates (T_o), s	**10**, 20, 40, 80, 160

scribed in Section 3.1. Serial isolates updates and queries using cell locks as follows. Before an object modification, each updater locks the current object's cell in exclusive mode. If the update is local, the object's position is modified, and the lock is released. If the update is non-local, the updater tries to lock the new (destination) cell as well. If the lock is obtained, the object is deleted from the old cell, inserted into the new cell, and both cell locks are released. If the lock on the new cell is not acquired, the update is restarted, as in PGrid. Query threads obtain cell locks in shared mode. After all cells overlapping a query range are locked, the query examines each cell in turn, releasing the locks one by one. No object-level locking is performed. Updates are facilitated via the secondary index, and queries are supported via the grid directory, as in the other variants.

For all of the compared multi-threaded indexes, an important issue is how to distribute the incoming workload to the index threads. A single queue for the messages becomes a bottleneck [28], as all index threads try to dequeue messages from it. To eliminate this possible source of contention from the experiments, the generated workload is distributed among the threads off-line so that each thread, independent from the other threads, can always obtain incoming messages without delay. The messages are assigned to threads in a round-robin fashion. Such a setting isolates all other sources of contention and exposes the thread-level parallelism available purely within the index structure and its operations. This makes the reported results independent from the chosen queue implementation and from how the load is distributed among the threads.

Workloads The indexes are studied in a range of experiments under massive workloads. The workloads are produced using an open-source moving object trace generator, MOTO[6] [7], that is based on Brinkhoff's moving-object generator [2]. MOTO follows a network-based object placement approach, where objects are placed and navigate (to a random destination) in a given road network. Table 3 shows the workload generator parameters and their values; the values in bold are default values. To obtain realistic skew and to stress test the indexing techniques, the generator was slightly modified so that half of the objects are placed in five major German cities according to the number of inhabitants in those cities. The queries are also distributed in those cities accordingly. This ensures that the most update-intensive regions are also the most queried ones.

5.2 Setting the Cloning Period in TwinGrid

For update-intensive workloads, stale query results and inefficient use of CPU cycles are the main problems of the indexing approaches, such as TwinGrid, that perform queries on periodically created snapshots of the indexed data. As PGrid provides fresh query results, we explore, first analytically and then experimentally, how to tune TwinGrid so that it returns sufficiently fresh query results to achieve a fair performance comparison with PGrid. First, we introduce the notion of query result *error rate*.

Definition 2. The error rate of a query result is the ratio of updates ignored by the query to the total number of objects inside the query's range. An update is ignored by a query if it occurs inside the query range before the query start time ($t_u < t_s$), but is not considered by the query.

By definition timeslice semantics (implemented by Serial) and the freshness semantics (implemented by PGrid) provide queries with zero error rate (see Sections 3.1 and 3.2). In snapshot-based approaches like TwinGrid, the error rate depends on how frequently a new snapshot is built. In the worst case, when a query is processed just before a new snapshot is created, the query misses all updates that occur within its range during the time between the two snapshots (the cloning period, T_{cp}, in TwinGrid).

To estimate the error rate in TwinGrid, we use the following notation. There are N_o moving objects in the monitored region. Given an average time between any two consecutive updates of an object, T_o, the number of updates per second in the region is $N_u = N_o/T_o$. Assuming that $T_{cp} < T_o$, we can compute the average error rate in TwinGrid as follows.

$$ER_{tg} = \frac{N_u}{N_o} \times \frac{T_{cp}}{2} = \frac{T_{cp}}{2T_o} \qquad (1)$$

Figure 4 illustrates the behavior of the above equation when varying the values of T_{cp} (the x axis) and T_o (separate lines). Even with update rates as high as $T_o = 1$ s[7], error rates below 1% are still feasible. However, this is at the cost of very high cloning rate. For example, to keep the error rate below 1% for objects sending updates every 10 seconds, 5 clonings per second are needed.

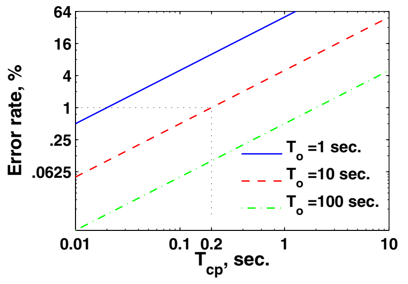


Figure 4: Error rate in TwinGrid, ER_{tg}.

Another property of existing snapshotting techniques is that a complete data snapshot is rebuilt every time from scratch [7, 28]. Even if only few objects get updated or few queries are issued, the CPU resources will be constantly and frequently consumed by the snapshotting. For example, keeping the error rate equal to 1% means that CPU resources are used for unnecessarily copying 99% of the data. In addition, while these calculations use average numbers, in realistic workloads, it is difficult to choose T_{cp} optimally, as update rates are highly skewed (peak hours versus night hours).

To empirically observe the effects of the cloning period on the error rate, throughput, and wasted CPU cycles, we performed experiments with TwinGrid using the default workload. The actual error rate is measured as follows. We randomly pick 1,000 queries

[6]Available at http://moto.sourceforge.net.

[7]E.g., if vehicles move at 10 m/s (36 km/h) and a high accuracy of 10 m is required, each car has to send an update every 1 s.

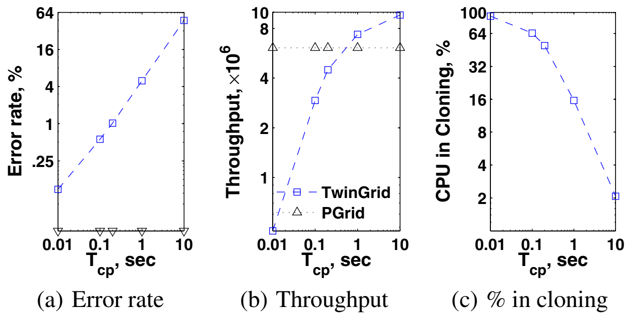


Figure 5: Effect of varying cloning period in TwinGrid.

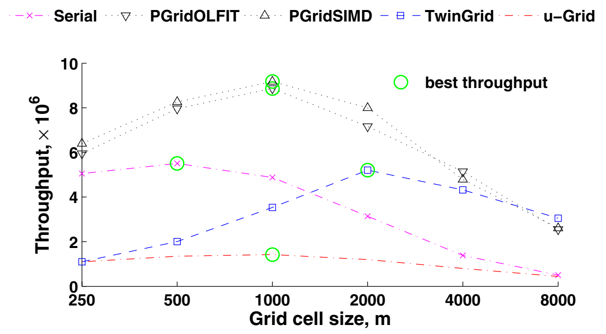


Figure 6: Optimal grid cell size (on Nehalem).

from the default workload and record their IDs. Then, during the experiments each time such a recorded query is processed, we store its result. The results reported by Serial serve as a baseline and are compared with the results of TwinGrid. The error rate is then:

$$\frac{N_{diff} + N_{fp} + N_{fn}}{N_s}\ , \qquad (2)$$

where N_{diff} is the number of objects with query-returned positions that are different from the corresponding positions returned by Serial, N_{fp} and N_{fn} is the numbers of false positives and false negatives, respectively, and N_s is the number of objects returned by Serial.

It is important to note that in these experiments and in all of the following experiments, to test the throughput-scalability of the indexes, the generated workloads are processed at maximum speed, that is, faster than they would be processed in the real setting when the processor is not utilized 100%. For TwinGrid, this speed up has no effect on the error rate and the percentage of the index-related CPU cycles spent on cloning, as clonings are sped up correspondingly. More specifically, the real-time cloning period T_{cp} is converted to an equivalent number of updates between clonings for the purpose of the experiments.

Figure 5 depicts how (a) the error rate, (b) throughput, and (c) percentage of CPU cycles spent on cloning changes when varying the cloning period from 10 ms, to 10 s and when objects report their positions every 10 seconds (T_o = 10 s.). As expected, and in accordance with the analytical study (Figure 4), the error rate of TwinGrid increases sharply and reaches 50% for a cloning period equal to 10 s. With a very frequent cloning, TwinGrid's throughput drops greatly (Figure 5(b)) as most of the time is spent in cloning rather than on processing (Figure 5(c)).

From the three figures, we can see that TwinGrid outperforms PGrid in terms of throughput when its cloning period is no less then ca. 0.6 s, delivering query results with an error rate of up to 4% and wasting at least 16% of its CPU cycles on cloning. To achieve a fair comparison with PGrid and Serial, we set TwinGrid's cloning period to guarantee the error rate no greater than 1% in the following experiments.

5.3 Experimental Findings

Varying the grid cell size and the bucket size Two important parameters effect the performance of all the indexes: the grid cell size and the bucket size. To obtain their optimal values, we exercise all indexes on each processor while trying different configurations. For example, we vary the grid cell size from 250 to 8,000 m. The results from the Nehalem processor are shown in Figure 6. Grid cell sizes are on the x-axis, while the throughput for both updates and queries is plotted on the y-axis.

In the single-threaded case, updates favor larger grid cell sizes (more updates are local), while queries favor grid cell sizes that approximate the average size of a query range. In the multi-threaded case, as the cell size increases so does the update-update interference and the update-query interference. TwinGrid favors cell sizes that are at least twice the size of those favored by the other indexes because its updates and queries are isolated. In contrast, since Serial suffers from conflicts due to extensive cell locking, it favors smaller cells. As expected, PGrid balances in-between the two and achieves the best throughput. The optimal grid cell sizes for each index are chosen as indicated by their best performance (circled markers in Figure 6). The optimal values are obtained similarly for the other processors (not shown). The bucket size has a relatively smaller impact on performance, but it was tuned in a similar manner.

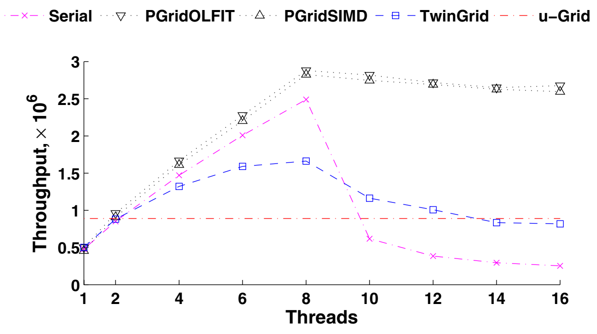
Multi-threaded scalability Figure 7 shows how the indexes scale versus the number of hardware threads on the different processors. The number of threads is on the x-axis, while the throughput of both updates and queries is plotted on the y-axis.

The more threads that are running on the same index structure, the more likely interference between different index operations becomes. Therefore, with each additional thread, the performance gain tends to decrease. All multi-threaded variants outperform the single-threaded u-Grid approach with already two threads. This implies that the overhead due to added thread-level synchronization is minor and would pay off already on a two-core machine.

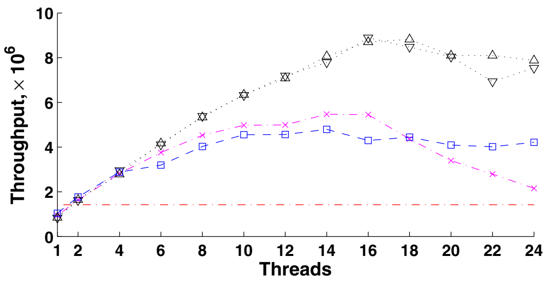
Serial is very competitive with up to 8 threads. Thus, on machines that support up to 8 hardware threads (Figures 7(b) and 7(e)) Serial might be a reasonable choice. However, with more threads, Serial fails to scale, as interference between update-update and update-query operations increases. Also, the number of software threads should not exceed the available hardware threads, as performance can quickly become worse than that of the single-threaded u-Grid (true on all platforms). This is because as soon as the hardware threads are exhausted, a latch-holding thread has a potential to be "parked" by the scheduler, forcing other threads to busy-wait without progress. This confirms Serial's sensitivity to the thread-level contention.

TwinGrid suffers from relatively rare update-update interference and outperforms Serial on the highly parallel T2 (Figure 7(d)). However, due to frequent cloning, TwinGrid often has to suspend and resume worker-threads so that a consistent data snapshot can be made. The more threads that are running, the more costly is to synchronize them. This continuous interruption of actual processing limits TwinGrid's throughput on all platforms.

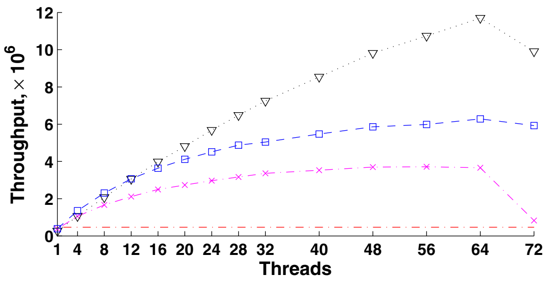
PGrid also experiences a relatively rare contention between update operations. Queries operate without long waiting latches, but



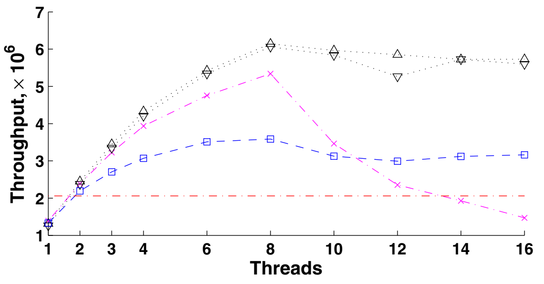
(b) Dual 4-core AMD Opteron 2350 2GHz (8 threads)



(c) Dual 4-core Intel Xeon X5550 2.67GHz (16 threads)

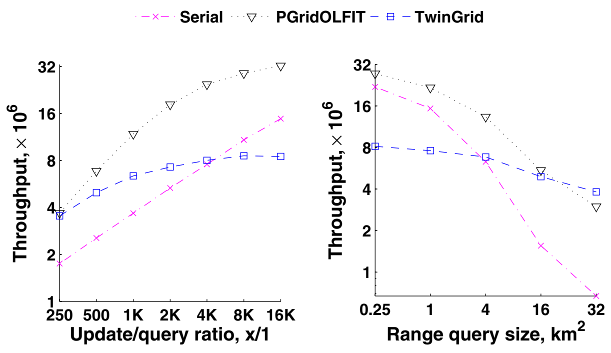


(d) 8-core Sun UltraSPARC-T2 1.2GHz (64 threads)

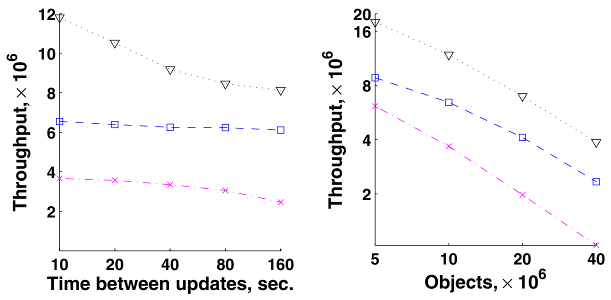


(e) 4-core Intel i7-2600 3.4GHz (8 threads)

Figure 7: Multi-threaded scalability.



(b) Varying update/query ratio (c) Varying range query size



(d) Varying T_o (e) Varying #objects

Figure 8: PGrid vs. TwinGrid vs. Serial (on T2).

require slightly more work on average, as it might be necessary to examine both current and logically deleted object versions. Nevertheless, PGrid scales near linearly on all platforms until all hardware threads are exhausted. Then, performance stabilizes or gets slightly worse, as software threads start competing for hardware threads.

There is no clear winner among the PGrid variants. On the Intel processors, the SIMDized version tends to perform better, while on the AMD processor, the OLFIT variant is slightly better. The Niagara T2 architecture supports just 64-bit wide SIMD registers; thus, only the OLFIT variant is used on T2. Nevertheless, we do not observe significant savings as in the previous micro-benchmark (see Table 1). This implies that a bigger fraction of cycles is consumed elsewhere.

In the following experiments, the number of threads is fixed at the number of available hardware threads on each processor. Also, since the same performance trends were observed on all of the processors, due to space limitations, we show the results only on one machine—the 8-core Sun Niagara T2 (1.2 GHz)—which has the most threads (8 per core, 64 in total).

Varying the updates/query ratio Figure 8(b) depicts the results when the updates/query ratio is varied from 250:1 to 16,000:1. The throughput of all indexes tends to increase as the number of long-running queries in the workloads decreases. Also, infrequent queries cause less interference with updates. Serial and PGrid benefit more from the increased updates/query ratio than does TwinGrid, where updates and queries do not interfere. PGrid achieves the best throughput. However, when querying more frequently than every 250 updates, it pays off to isolate update and query processing as in TwinGrid, whereas update-only workloads are more efficient on Serial (no need to maintain logically deleted object copies).

Varying the range query size The impact of varying the query size is shown in Figure 8(c). Larger queries need to inspect more cells, buckets, and objects, causing decreasing performance for all indexes. Serial is affected the most, as the larger the queried regions, the more likely it is that parallel updates hit the query range and form hotspots. Since TwinGrid operates on two separate data copies, it is affected least. PGrid offers the best performance, and TwinGrid should be considered only for workloads with very large queries (more than 16 km^2).

Varying T_o The target applications exhibit update locality—the next location reported by an object is likely to be close to the previously reported one. This property is exploited by the indexes, as local updates are simply processed by overwriting the outdated coordinates. Non-local updates require more processing. Serial and TwinGrid require extra locking while an object is being deleted from its old cell and inserted into its new one. PGrid creates a logically deleted copy of the object for each non-local update, and it deletes it at the next update.

To measure this effect, we vary the average time between two consecutive updates of an object (T_o). The larger this time, the greater the likely deviation of an object from its previous position becomes. Figure 8(d) shows the results. All indexes are affected negatively, but with different impact. Remember that the indexes are configured with different grid cell sizes (Figure 6), so they are subjected to different amounts of non-local updates. With T_o varying from 10 to 160 s, we estimate that the fraction of local updates drops from 88% to 34%, from 77% to 20%, and from 94% to 56% in PGrid, Serial, and TwinGrid, respectively. Therefore, TwinGrid is the least affected, as more than half of the updates remain local (and non-local updates do not interfere with queries). Nevertheless, PGrid still achieves the best throughput even when one third of its updates are non-local (when T_o=160 s).

Varying the number of objects As the number of objects increases, the updates and queries operate on an increasingly large data set, which adversely affects performance for all indexes (Figure 8(e)). Also, TwinGrid favors smaller index sizes, as less data has to be copied during the cloning phase. With a small database size, TwinGrid's performance becomes similar to that of Serial. The reason might be that the constant cloning in TwinGrid prevents it from exploiting cache-sharing opportunities, as all level caches are flushed. With a large database, cache-sharing becomes difficult for any index. Nevertheless, PGrid stays on top for all the considered data sizes.

6. CONCLUSIONS

Today, increased on-chip parallelism is a key mean of improving processor performance. This development calls for software techniques that are capable of scaling linearly with the available hardware threads. Moving-object workloads with queries and massive numbers of updates render it particularly challenging to avoid inter-thread interference and thus achieve scalability.

We present a parallel, grid-based indexing technique for moving-object workloads. Key advantages of our technique include up-to-date query results and the ability to make efficient use of thread-level parallelism. The technique improves on existing, snapshot-based approaches that suffer from stale query results and waste CPU resources on frequent snapshotting. An empirical study with four quite different, modern processors shows that the proposed technique is capable of outperforming existing alternatives and that it scales near-linearly with the number of available hardware threads.

7. ACKNOWLEDGMENTS

This research was supported by grant 09-064218/FTP from the Danish Council for Independent Research | Technology and Production Sciences. We thank Kenneth A. Ross for his support with the experimental hardware. We also thank the anonymous peer reviewers for their constructive feedback.

8. REFERENCES

[1] V. Akman, W. R. Franklin, M. Kankanhalli, and C. Narayanaswami. Geometric computing and the uniform grid data technique. *Computer Aided Design*, 21(7):410–420, 1989.

[2] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6:153–180, 2002.

[3] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, pp. 181–190, 2001.

[4] K. Chakrabarti and S. Mehrotra. Efficient concurrency control in multidimensional access methods. In *SIGMOD*, pp. 25–36, 1999.

[5] S. Chen, C. S. Jensen, and D. Lin. A benchmark for evaluating moving object indexes. *PVLDB*, 1(2):1574–1585, 2008.

[6] A. Civilis, C. S. Jensen, and S. Pakalnis. Techniques for efficient road-network-based tracking of moving objects. *IEEE TKDE*, 17(5):698–712, 2005.

[7] J. Dittrich, L. Blunschi, and M. A. V. Salles. Indexing moving objects using short-lived throwaway indexes. In *SSTD*, pp. 189–207, 2009.

[8] J. Gray and A. Reuter. Transaction processing: concepts and techniques. *Morgan Kaufmann Publishers*, 1993.

[9] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks and degrees of consistency in a shared data base. In *VLDB*, pp. 428–451, 1975.

[10] S. Hwang, K. Kwon, S. Cha, and B. Lee. Performance evaluation of main-memory R-tree variants. In *SSTD*, pp. 10–27, 2003.

[11] Intel 64 and IA-32 Architectures Software Developer's Manual. *Volume 3A: System Programming Guide, Part 1*. Order number: 253668-037US, January 2011.

[12] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B^+-tree based indexing of moving objects. In *VLDB*, pp. 768–779, 2004.

[13] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *SIGMOD*, pp. 139–150, 2001.

[14] M. Kornacker and D. Banks. High-concurrency locking in r-trees. In *VLDB*, pp. 134–145, 1995.

[15] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and recovery in generalized search trees. In *SIGMOD*, pp. 62–72, 1997.

[16] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in R-trees: a bottom-up approach. In *VLDB*, pp. 608–619, 2003.

[17] P. L. Lehman and s. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM TODS*, 6(4):650–670, 1981.

[18] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. *PACT*, 0:261–270, 2009.

[19] V. Ng and T. Kameda. Concurrent access to R-trees. In *SSD*, pp. 142–161, 1993.

[20] L.-V. Nguyen-Dinh, W. G. Aref, and M. F. Mokbel. Spatio-temporal access methods: Part 2 (2003 - 2010). *IEEE DEB*, 33(2):46–55, 2010.

[21] J. M. Patel, Y. Chen, and V. P. Chakka. Stripes: an efficient index for predicted trajectories. In *SIGMOD*, pp. 635–646, 2004.

[22] POSIX.1-2008. The open group base specifications. Also published as IEEE Std 1003.1-2008, July 2008.

[23] R. Rastogi, S. Seshadri, P. Bohannon, D. W. Leinbaugh, A. Silberschatz, and S. Sudarshan. Logical and physical versioning in main memory databases. In *VLDB*, pp. 86–95, 1997.

[24] B. Salzberg. Grid file concurrency. *Information Systems*, 11(3):235–244, 1986.

[25] S. I. Song, Y. H. Kim, and J. S. Yoo. An enhanced concurrency control scheme for multidimensional index structures. *IEEE TKDE*, 16(1):97–111, 2004.

[26] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pp. 1150–1160, 2007.

[27] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: an optimized spatio-temporal access method for predictive queries. In *VLDB*, pp. 790–801, 2003.

[28] D. Šidlauskas, K. A. Ross, C. S. Jensen, and S. Šaltenis. Thread-level parallel indexing of update intensive moving-object workloads. In *SSTD*, pp. 186–204, 2011.

[29] D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen, and D. Šaulys. Trees or grids? Indexing moving objects in main memory. In *GIS*, pp. 236–245, 2009.

[30] M. Yiu, Y. Tao, and N. Mamoulis. The B^{dual}-tree: indexing moving objects by space filling curves in the dual space. *PVLDB*, 17(3):379–400, 2008.