The VLDB Journal manuscript No.
(will be inserted by the editor)

Processing of Extreme Moving-Object Update and Query Workloads in Main Memory

Darius Šidlauskas · Simonas Šaltenis · Christian S. Jensen

Received: date / Accepted: date

Abstract The efficient processing of workloads that interleave moving-object updates and queries is challenging. In addition to the conflicting needs for update-efficient versus query-efficient data structures, the increasing parallel capabilities of multi-core processors yield challenges. To prevent concurrency anomalies and to ensure correct system behavior, conflicting update and query operations must be serialized. In this setting, it is a key concern to avoid that operations are blocked, which leaves processing cores idle.

To enable efficient processing, we first examine concurrency degrees from traditional transaction processing in the context of our target domain and propose new semantics that enable a high degree of parallelism and ensure up-to-date query results. We define the new semantics for range and k-nearest neighbor queries. Then we present a main-memory indexing technique called PGrid that implements the proposed semantics as well as two other variants supporting different semantics. This enables us to quantify the effects different degrees of consistency have on performance. We also present an alternative time-partitioning approach. Empirical studies with the above and three existing proposals conducted on modern processors show that our proposals scale near-linearly with the number of hardware threads and thus are able to benefit from increasing on-chip parallelism.

D. Šidlauskas
Department of Computer Science, Aarhus University,
Aarhus, Denmark
E-mail: dariuss@cs.au.dk

S. Šaltenis
Department of Computer Science, Aalborg University,
Aalborg, Denmark
E-mail: simas@cs.aau.dk

Christian S. Jensen
Department of Computer Science, Aalborg University,
Aalborg, Denmark
E-mail: csj@cs.aau.dk

Keywords Spatio-temporal indexing · Concurrency · Thread-level parallelism

1 Introduction

We are witnessing a rapid growth in Internet-worked, geo-positioned smartphones and other mobile devices. Likewise, location-related services that target the users of such devices are rapidly growing. Consequently, cost-effective server-side infrastructures are needed that are capable of supporting massive, location-related update and query workloads generated by very large populations of such 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.

Another, more futuristic application scenario relates to autonomous vehicles, where an in-vehicle system continually processes data from sensors, including a GPS, video cameras, a laser array, and other radars mounted around the vehicle. The system monitors surrounding moving objects (obstacles) and take appropriate actions, e.g., stopping the car before it hits a pedestrian. Like in the previous scenario, the processing loads are massive. Unlike in the previous scenario, there are no humans in the loop and sensitivity to latency is high.

The workloads we consider cannot be sustained by disk-based techniques in a centralized setting, but would need a costly infrastructure with very large numbers of disks. Motivated by two observations, we instead investigate main-memory indexing. First, the size of the raw location data is relatively small, e.g., 8 bytes [11]. Second, continuously dropping prices and increasing capacities of RAM chips enable single server machines that offer main memory storage in the TB range. As a result, it is feasible to manage billions of moving objects in main memory on a single machine. In case of failures where the data in main memory is lost, we rely on the high object update rate to quickly populate the database with an up-to-date state rather than populating the database from a backed up database.

The main challenge with such update- and query-mixed workloads is to harness the parallel processing capabilities of current chip multi-processors (CMPs). When read and write operations are processed in parallel, an appropriate concurrency control scheme is needed to maintain a consistent database state and ensure correct query results. Optimistic concurrency control methods anticipate that conflicts between transactions will not occur either because write operations are rare or because write and read operations access different data items. In the application that we target, frequently queried objects are also updated frequently. On the other hand, concurrency control based on locking can cause contention. Query threads must wait for update threads holding exclusive locks to avoid reading inconsistent states. For the same reason, update threads must wait for query threads holding shared locks. Long-duration queries have the potential to slow down rapid single-object updates. Moreover, extra care must be taken to avoid other concurrency anomalies such as deadlocks and phantoms [15]. To avoid deadlocks, some operations are often aborted, while to avoid phantoms, predicate or range locks are often used. All of this combines to render it challenging to exploit the opportunities for parallelism.

The paper studies parallelism at two levels: semantic and implementation. At the semantic level, it investigates update and query isolation requirements applicable to the processing of moving object workloads. It shows how choosing the right level of relaxed serializability of traditional database management systems enables more parallelism while preserving the query semantics for the targeted applications. The paper proposes freshness semantics that yield more up-to-date query results (relevant for continuously changing data) and enable a high degree of parallelism. The semantics are defined for two types of queries—range and k nearest neighbor—that are both studied widely in the literature and used in practice.

At the implementation level, the proposed algorithms are designed carefully so that locking is done on as little data as possible and for as short time as possible. The proposed index structure called PGrid (parallel grid) employs light-weight locking (e.g., 1-byte latches with the atomic CAS instruction are used instead of 40-byte `pthread` mutexes) that is used only by update threads; query threads are never blocked. Update locks have small coverage and short duration due to PGrid's fixed and uniform grid [1] that avoids expensive structure modifications, obviating the need to lock entire sub-trees or grid cells, as in adaptive approaches [24, 30, 36].

Non-blocking queries are possible in PGrid due to its multi-version concurrency control scheme that enables queries to read previous object versions. The previous versions 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.

The paper reports on an extensive empirical study that encompasses four diverse multi-core platforms. The study shows that PGrid scales near-linearly with the number of hardware threads and is capable of outperforming existing alternatives. This includes outperforming the previous state-of-the-art snapshot-based approach that trades query freshness for update and query performance [42]. PGrid offers three main advantages over snapshot-based techniques: (i) it provides up-to-date query results; (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 multi-version concurrency control scheme treats updates as atomic operations rather than as combinations of deletions and insertions, guaranteeing query semantics with no phantom objects.

Additionally, the study compares PGrid with two alternatives that enable the same degree of parallelism, but offer different query semantics. First, PGrid is compared against a variant that supports serializable query semantics. This variant has up to 50% lower throughput than PGrid due to extra (unrelated to contention) computation. Second, PGrid is compared against a time-partitioning approach that increases performance at the cost of both reduced query freshness and increased latency (or delay time). This approach, called TP-Grid, achieves up to 25% higher throughput, but might delay query processing and generally returns slightly outdated results.

The paper is an extension of a conference paper [44] that introduced and evaluated PGrid on four different multi-core processors. The paper substantially extends the conference paper with five main contributions. First, it investigates query semantics in terms of degrees of consistency from traditional database management systems. The different semantics are placed in the context of known concurrency anomalies [16] to provide guarantees for query result

when moving-object updates and queries are processed in parallel. Second, in addition to range queries, it proposes freshness semantics for k nearest neighbor queries. Doing so is not trivial since the range to be scanned by a k nearest neighbor query is not fixed and depends on the continuously changing positions of objects around the query point. Third, it presents a variant of PGrid that supports serializable (timeslice) semantics. This enables us to quantify the costs of serializable execution (or the costs of a higher degree of consistency) in PGrid. Fourth, the time-partitioning approach is new. Lastly, the paper replaces the conference paper's empirical studies with studies that utilize newer (more parallel) hardware, taking into account all the 8 indexing techniques considered in the paper, and it offers new insight into the scalability of the techniques with continuously increasing parallelism.

The remainder of the paper is organized as follows. Section 2 covers preliminaries and the problem setting. Section 3 describes formally the different query semantics supported by the proposed indexing techniques. The PGrid indexing technique and the accompanying algorithms are described in Section 4, while alternatives are described in Section 5. Empirical studies of the proposals are covered in Section 6. Section 7 covers related work, and Section 8 concludes the paper.

2 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 supports the delivery of a variety of location-based services. 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.

An update message includes the object's id (oid), its new two-dimensional coordinates (x,y), and an update timestamp (t_u). Albeit 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_{low},y_{low}) and (x_{high},y_{high}). A k nearest neighbor (kNN) query is defined by its query point and a number (k) of closest points required. kNN queries can be derived from range queries [19]. Queries examine many objects and take much longer to process than do updates.

The size of the raw location data is relatively small, e.g., a moving object representation including the object's identifier, two-dimensional position, and speed vector can be packed into 8 bytes [12]. In our setting, a moving object is represented by a 16-byte tuple (oid,x,y,t_u), implying that 64M moving objects (more than all registered vehicles in Germany [25]) require just 1GB of storage. Therefore, with current RAM chip capacities up to several TB per server, single machines can store billions of moving objects in main memory.

It is non-trivial to provide accuracy guarantees concerning the known position of a moving object, as the object's position changes continuously. We assume that updates occur according to the shared-prediction-based protocol [10, 45], which ensures that an object's position as known by the server is no further away from its actual (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 most recently reported position. Figure 1 illustrates an example with five moving objects. The black dots are the object positions stored in the database, while the white dots are the actual positions. Dashed circles (with radius δ) around the stored positions indicate the possible object locations. Consider the range query represented by the solid-line rectangle, which reports objects A, B, and C. Compared to the actual object positions, the query has one false positive (C) and one false negative (D).

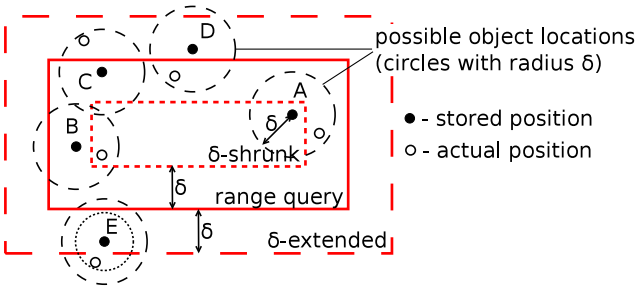


Fig. 1: Effects of extending/shrinking query range.

Notably, with the assumed tracking protocol, the uncertainty is only around the borders of the query range. To report all objects that might be in a query range, the range must be extended by δ[1]. However, this might increase the number of false positives. In the example, the δ-extended query, represented by the dashed rectangle, does not report any false negatives (D is included), but the number of false positives increases to two (E is included). If instead we want to capture all objects that are guaranteed to be in a query range, the range must be shrunk by δ. This might increase the number of false negatives. In the example, the δ-shrunk query, drawn using a dotted line, does not report any false positives (C is excluded), but the number of false negatives increases to two (B is excluded). We assume that all incoming queries are already extended, shrunk, or left unmodified according to user requirements.

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

Consistent with other studies [11,19,33,39,48], we also assume that we know (possibly conservatively) the maximum possible speed of any object, v_{max}, e.g., 50 m/s [19,33] or 60 m/s [11]. Together with the δ threshold, the minimum time between consecutive updates of an object then becomes $T_o = \delta / v_{max}$.

Recording the last-update timestamps (t_u) of objects in an index enable more refined query results when combined with the maximum speed (v_{max}) assumption . First, the δ-extended query is performed to ensure that there are no false negatives. Then, for each reported object, we construct the possible region in which the object can be. This is the circle with radius $min(v_{max} \times (t_q - t_u), \delta)$ centered at the query-observed database position of the object, where t_q is the query time. The objects whose possible region does not intersect with the query range are filtered out. In the example, the dotted circle around object E indicates its possible region at time t_q. The region does not intersect with the original query rectangle and thus can be eliminated. Moreover, one can calculate intersection probabilities for objects based on the overlap between such possible regions and the query rectangle, upon which it is possible to discard objects with probabilities that do not meet a given probability threshold. Recently updated objects have especially small circles and can be removed or left in the result with high certainty. We assume that such query pruning is done in a post-query phase according to user requirements.

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.

3 Semantics and Parallelism

We proceed to describe in detail the query semantics supported by the proposed indexing techniques. In a single-threaded execution, at most a single query or update is being processed at any point in time. In a multi-threaded setting, system throughput can be increased by means of *intra-operation* and *inter-operation* parallelism. Since we are faced with simple single-object update operations, our focus is on inter-operation parallelism, where different operations are processed in parallel.

When executing queries and updates in parallel, the question of query semantics arises naturally: how is the correct result of a query defined? We describe correctness properties of queries in terms of the degrees of consistency known from conventional database management systems [15]. In the following subsections, unless noted otherwise, an update is a transaction consisting of a deletion of an object's old position and an insertion of the object's new position. When objects enter or leave the system, an update can also be a single insertion or a single deletion. A query is a transaction consisting of read-only operations.

Serializable execution of operations is a desirable property, but query semantics corresponding to lower levels of isolation among transactions enable more parallelism. In the following subsections, we discuss different semantics and argue that slightly relaxed semantics are acceptable for the applications we consider and increase the potential parallelism significantly. The findings in this section are general in nature and are not restricted to a particular indexing technique.

3.1 Serializability (Full Isolation)

A concurrency control (CC) scheme that ensures complete isolation among update and query operations can be implemented as follows. An update transaction, operating on a single object, obtains an exclusive lock on that object. The lock is released as soon as the update completes. A query transaction obtains shared lock(s) on the range(s) of the data space that must be accessed to produce its answer [15]. 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 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. As illustrated in Figure 2, despite available resources on multi-core CPUs, rapid updates can be delayed by the time it takes to process a prior long-running query.

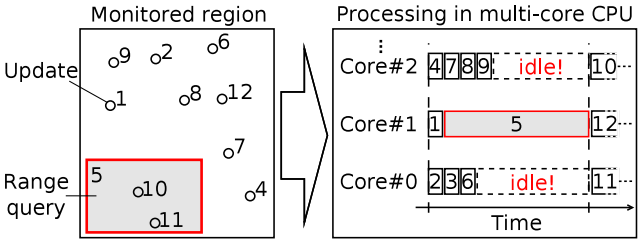


Fig. 2: Serializable processing: a long-running range query blocks rapid updates. Numbers indicate arrival order.

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 [24,30]; and in grid-based approaches, entire cells or buckets (data pages) can be locked [36]. This results in even

more update/query interference. Past studies [9, 17] suggest that workloads with both queries and updates do not scale in such settings: only few cores are utilized efficiently.

Using the above locking protocol, update and query transactions cover their actions by locks (are well-formed), and as no unlock precedes a lock, they use two-phase locking. Thus, based on the fundamental serialization theorem [15], any parallel execution of the transactions is serializable, i.e., is equivalent to some serial execution of the transactions.

In a serial execution, a query result reflects all previously executed transactions and none of the transactions executed after the query. We say that this corresponds to *timeslice* query semantics because a query result is based on the objects positions valid at some specific time point in some serial ordering of the transactions, which is usually similar to the arrival order of the transactions. With the above scheme, the specific time point corresponds to the query processing start time when the necessary locks are acquired and the subsequent conflicting updates must wait.

Such lock-based isolation among transactions yields degree 3 isolation, which is the highest level of consistency in database management systems [16]. However, as suggested in Figure 2, this level of consistency limits concurrency. Consider the traffic monitoring scenario. In a single-threaded setting, an update takes circa 1 microsecond, while a query might take up to a few milliseconds depending on the size of the result [43]. This implies that an update might be delayed by three orders of magnitude longer than its actual processing time when it hits a region locked by a query.

3.2 Snapshot Isolation

The following multi-version concurrency control (MVCC) scheme ensures *snapshot* isolation [4], where all reads by a transaction see a consistent snapshot of the database as of transaction start and where this snapshot remains unaffected by writes of other concurrent transactions. At transaction start, a transaction obtains a timestamp t_x that is larger than any existing timestamp given to a transaction. An update transaction, operating on a single object, obtains a lock on that object. Before releasing the lock, the update creates a new copy (version) of the object with its new position and tags it with its timestamp, t_x. New object versions created by other concurrent transactions with timestamps that exceed t_x are invisible to the transaction. A query transaction operates without locks and can safely read object versions with timestamps lower than its own. Outdated object versions are removed when no active transactions exist that need them. With this scheme, each moving object can have multiple versions in the system at any time.

However, snapshot isolation is not serializable and is vulnerable to several concurrency anomalies because of *constraint violations*, e.g., the write skew anomaly [3] and the read-only transaction anomaly [14]. For example, if there is a constraint to maintain the uniqueness of some object positions, snapshot isolation cannot guarantee this (as concurrent transactions do not see each others updates). The applications we target do not have such multi-object constraints: update transactions are limited to single data items. The incoming (individual) moving object positions are independent of each other and are always accepted as reported. Therefore, moving-object update and query transaction executions satisfying snapshot isolation give queries the same timeslice semantics as with serializability. Similarly, the TPC-C benchmark application executes under snapshot isolation without serialization anomalies [13].

The major advantage of snapshot isolation is that it eliminates all interference between updates and queries present in serializable mode (Section 3.1). Queries can access the entire database without being affected by writes made by concurrent updates. As a result, much more parallelism is achieved. In fact, contention-related performance degradation is at the same level as is degree 0 isolation [3]. Updates still might interfere with each other, but single-object locks are short.

Performance degradation unrelated to contention is worse because of the need to maintain multiple object versions: updates need to create new versions, while queries need to decide which version to read. Studies of an existing MVCC scheme for main memory indexing show that this cost can be excessive, rendering the scheme suitable only for query intensive workloads [35]. This is mainly due to concurrent memory allocations/de-allocations that cause significant overhead in terms of operating system calls (`malloc`, `free`, etc.). Subsequent work [7] reduces this overhead by creating a memory pool for each processor in the system.

Note that timeslice query semantics under snapshot isolation guarantees fresh query results (as with serializability). At any time, there might be several outdated copies of an object in the system, but a query examines only the most recent version of an object as of the start of the query.

3.3 Two-Snapshot Isolation

Another way to reduce the versioning overhead is to limit the number of versions maintained. Consider an approach where each object has only two versions: the previous and the current. Query transactions are directed to the previous versions that are read-only. Update transactions operate on the current versions that are write-only. The previous versions are refreshed regularly based on the current versions so that query results are reasonably up-to-date. During a refreshing phase, updates are suspended so that the new read-only versions correspond to a consistent database state as of some time point. Since the degree of parallelism is the same

as for snapshot isolation, the approach scales well on multi-core platforms [11, 42].

However, as before, this scheme does not yield serializable semantics, and it additionally suffers from outdated query results. No matter when a query is issued, its result is based on the database snapshot created during the last refreshing phase. We thus term the supported query semantics *stale-timeslice* semantics. To enable a fair comparison with timeslice query semantics, we introduce the notion of query *staleness*:

Definition 1 The staleness of a query is the ratio of update transactions ignored by the query transaction to the total number of objects in the system. An update transaction is ignored if it has a lower timestamp than the query transaction, but is not taken into account by the query.

The query staleness under stale-timeslice semantics depends on a tuning parameter that controls the snapshotting frequency, F_s. In main memory indexing, very frequent snapshotting, on the order of tens of snapshots per second, is feasible [42]. However, frequent snapshotting implies a substantial and unattractive waste of computing resources. For example, if one aims to maintain a maximum query staleness below 1%, snapshotting must be made whenever 1% of all objects are updated. The versions created for the remaining 99% of objects are superfluous. Appendix A details the trade-off between query freshness and staleness.

Given the above waste, incremental snapshotting techniques might sound like an attractive approach. However, recent work [41] shows that incremental snapshotting is often more expensive than brute-force (all data) snapshotting in update-intensive applications.

Another drawback of two-snapshot isolation is that it introduces query *latency*. If a query is received during the snapshotting phase, its processing is delayed until the phase completes. Given that queries already suffer from stale results, stale-timeslice semantics might limit the target applications significantly.

3.4 Freshness Isolation

We proceed to consider a CC scheme that permits an anomaly known as data *phantoms* [15]. This scheme differs from the one defined in Section 3.1 in that query transactions do not use range locking. That is, update transactions are allowed in a region of a concurrent query transaction. Assume that to prevent partial reads and writes of object data, the CC scheme is modified to ensure the atomicity of updates (deletion-insertion pairs). We aim to understand which guarantees this scheme sacrifices, when compared to timeslice semantics.

Assuming that a query transaction lasts from t_s to t_e, Figure 3 shows a snapshot at some point in time between t_1 and t_2 ($t_1, t_2 \in [t_s, t_e]$) of a range query occurring simultaneously with several updates. At t_1, the query has already inspected half of its range (the grey region), and processing is in progress (striped region). The black dots are object positions at the beginning of the query (at t_s), and the white dots are their updated positions due to updates that occurred during $[t_1, t_2]$. We can identify four inconsistencies in this simple CC scheme when compared to a scheme offering timeslice semantics:

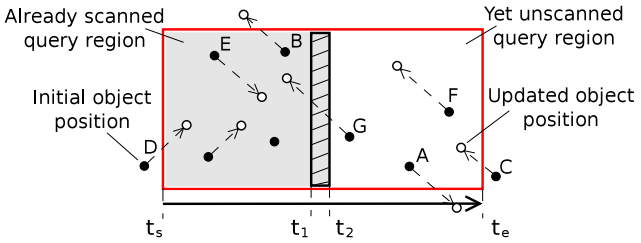


Fig. 3: Parallel updating and querying.

(i1) Object A is in the query range at t_s. However, it exits 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 the range during $[t_1, t_2]$, but is captured in both CC schemes.

(i2) Object C is not within the query range at t_s. However, it is reported because 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 either CC scheme.

(i3) Some of the reported object positions are fresher than others. For example, objects E and F are both in the query range before and after being updated. 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 yet unscanned to the already scanned query region.

We argue that cases i1, i2, and i3 can be tolerated easily in the targeted application domains. In fact, the behavior in these cases may often be preferred in the context of continuously changing data because freshness of results is preferred over returning results that were consistent (according to 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 make use of the assumption 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. Doing so guarantees that a query always encounters at least one object version and so does not miss any objects. In Figure 3, the query then reports G's previous position (black dot).

We call the level of guarantees that corresponds to allowing cases i1 to i3, but not 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 for range query 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 2 Given a range 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 hold:

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 or not o is in the query result is determined by its up-to-date position. The second part deals with objects that are updated during the query processing and covers the cases discussed already. Observe that cases 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 current pos° might be inside or outside the query range).

Concurrent transaction executions allowed by Definition 2 are not guaranteed to be serializable. For example, assume that two transactions are about to update objects A and B while two queries $\Re_1$ and $\Re_2$ are running concurrently. What happens when both updates are inside the ranges of both queries? According to Definition 2, $\Re_1$ might take into account only the update of A, while $\Re_2$ might take into account only the update of B. As a result, such a parallel execution is generally not serializable. Figure 4 depicts two possible parallel executions allowed by freshness semantics: one serializable and one not serializable.

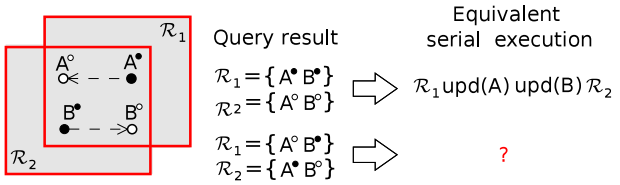


Fig. 4: Freshness semantics is not serializable.

Following the same ideas, we define the freshness semantics for the kNN query. The key difference between a range query and a kNN query is that whether a given object qualifies for the kNN query depends not only on the query and the current position of the object, but also on the positions of other objects. Thus, although the object's position may be updated once or not at all during the query processing, the object's belonging to the query result may change multiple times.

We define $o.pos(t)$ to be the position of the object o as recorded in the database at time t. In particular, for $t \in [t_s, t_e]$, $o.pos(t) = o.pos^\bullet$ if $t < o.t_u$, and $o.pos(t) = o.pos^\circ$ if $t \geq o.t_u$. Also, let $X_{(k)}$ denote the k-th smallest value in a set X of values.

Definition 3 Given a set of objects S and a kNN query at point q with processing time $[t_s, t_e]$, its result $O \subseteq S$ is said to satisfy freshness semantics if for any object o, the following hold:

1) if $\max_{t\in[t_s,t_e]} dist(q, o.pos(t)) < \{\min_{t\in[t_s,t_e]} dist(q, x.pos(t)) \,|\, x \in S\}_{(k)}$ then $o \in O$
2) if $\min_{t\in[t_s,t_e]} dist(q, o.pos(t)) > \{\max_{t\in[t_s,t_e]} dist(q, x.pos(t)) \,|\, x \in S\}_{(k)}$ then $o \notin O$
3) otherwise o may or may not belong to O

Case 1) is the core of the definition and corresponds to case 2.a), as well as a part of case 1), in Definition 2. The idea is that for each object from the dataset at least one position valid during some part of $[t_s, t_e]$ should be considered when constructing the answer of a kNN query. In the worst case, the position of o that is furthest away from q is considered (the left side of the inequality). The object o should definitely be in the answer if that position is closer to q than the k-th closest object in the best-case answer where the closest

positions of all objects are considered (the right side of the inequality).

Figure 5 shows the distances of a set of objects from the query point during the processing of a 3NN query. Solid lines represent object positions read by the query while dashed lines show missed object positions. Object D is covered by case 1) of the definition and should be always returned. Note that case 1) also includes objects that did not update during $[t_s, t_e]$, such as object E in the example.

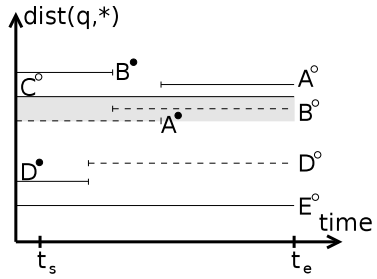


Fig. 5: 3NN query with the result $\{E^\circ, D^\bullet, C^\circ\}$, satisfying the freshness semantics.

Case 2) is the opposite of case 1), stating that an object should definitely be excluded from the answer if the position of o closest to q is further away than the k-th closest object in the worst-case answer constructed considering the furthest positions of all objects.

Case 3) states that, due to concurrent updates, we cannot guarantee the accuracy of the result concerning the objects that are on the border of the "query range," i.e., not covered by cases 1) and 2). In contrast to range queries, this also affects the objects that did not update during $[t_s, t_e]$.

In Figure 5, object C was never among the three nearest neighbors of q during $[t_s, t_e]$. In this example, it is falsely reported as there were times (the whole interval $[t_s, t_e]$ in the example) when it was in-between the 3rd object of the best-case answer $\{E^\circ, D^\bullet, A^\bullet\}$ and the 3rd object of the worst-case answer $\{E^\circ, D^\circ, C^\circ\}$. Note that the distances from q to the best-case k-th object and to the worst-case k-th object can differ by at most δ, the distance between any two consecutive updates of an object (cf. Section 2). This difference is shown as the gray region in Figure 5.

Whether C, or any other object crossing the gray region, is reported or not depends on the order in which object positions are encountered by a query algorithm. An interesting way to interpret this uncertainty "on the border" is to compare it to the indeterminism of reporting only a random subset of objects whose distance to q is exactly equal ($\delta = 0$) to the distance from q to the k-th nearest neighbor, when there are too many such objects and exactly k objects have to be reported. Allowing one (potentially missed) update per object during $[t_s, t_e]$, the uncertain border of the query is widened from 0 to at most δ.

Finally, observe that case 1) of the definition is always satisfied by strictly less than k objects. Let $k'(0 \leq k' < k)$ be the number of objects satisfying case 1). While not explicitly required by the definition, a query algorithm should preferably choose the remaining $k - k'$ objects with the smallest distances to q (as seen by the algorithm) among the objects satisfying case 3).

3.5 Summary

Freshness isolation enables queries to report more up-to-date data items when possible without the risk of missing an item. In terms of database consistency [3], freshness isolation situates between degree 2 and snapshot isolation (Figure 6). It is strictly less restrictive than snapshot isolation because it allows some data phantoms (i.e., i1–i3), which snapshot isolation does not. It is strictly more restrictive than degree 2 isolation because degree 2 isolation allows all data phantoms, while freshness isolation prevents i4.

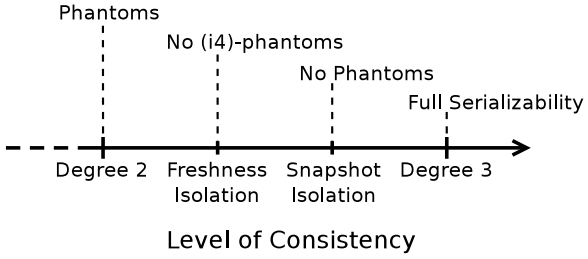


Fig. 6: Semantic relationships.

We shall later see that the degree of parallelism in freshness isolation is comparable to that in snapshot isolation (and degree 2 isolation), while computation unrelated to contention is lower than in snapshot isolation. This is because updates and queries do not interfere with each other, while only necessary tuples are replicated.

4 Parallel Grid

Here, we detail the *PGrid* indexing technique that implements freshness semantics for range and kNN queries. First, we outline the index structure, the operations supported, and the types of locks used. Next, we cover the update and range and kNN 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 [43]. Queries are serviced using a uniform grid [1], while updates are facilitated via a secondary index in bottom-up fashion [26]. Figure 7 depicts PGrid's components and their structure.

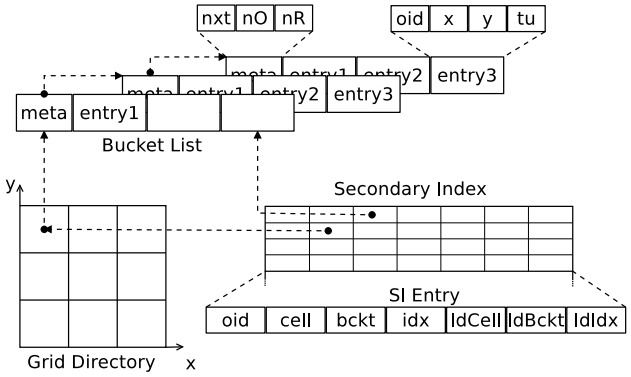


Fig. 7: PGrid structure.

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, the object data that belongs to a cell is stored in a linked list 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 belong to 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, to be explained later.

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*) provide 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 [43], 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. There is no need for explicit object deletion and insertion operations during local updates, and 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 a cell yet to be scanned by a query to a cell that has already been scanned (see Figure 3).

To handle non-local updates, PGrid introduces a notion of *logical deletion*. A non-local update 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 on the next update of the object. 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 the other. To guarantee consistency between the two, PGrid's concurrency control includes two types of locking: object locking and cell locking. The locks are independent of 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 (write) access just to one particular object, it has only 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 the new bucket becomes the first. Cell locks are required only during physical object insertions and deletions. As detailed in the following, both operations (deletion of *old* and insertion of *new* positions) 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

Algorithm 1 shows the pseudo-code of PGrid's update algorithm. The algorithm takes as 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 18). After a secondary index lookup, 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.

Algorithm 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     sie.ldCell = sie.cell;
14     sie.ldBckt = sie.bckt;
15     sie.ldIdx = sie.idx;
16     insert(new, newCell, sie);                      // physical insertion
17     obj.tu = -new.tu;                               // mark as logically deleted
18 unlockObj(new.oid);
```

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 entry (*sie*) are non-null, and the tuple has to be (physically) deleted (Line 7). As will 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 (Line 16) and the outdated tuple is logically deleted. A logical deletion is carried out in two parts. First, before the insertion (in Lines 13–15), the fields in *sie* referring to the current object tuple (*cell*, *bckt*, *idx*) are copied over the fields referring to the logically deleted tuple (*ldCell*, *ldBckt*, *ldIdx*). Second, after the insertion (Line 17), the timestamp of the outdated tuple is set to the current time (*new.tu*). 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.

Algorithm 2 contains the pseudo-code for physical object deletion. 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 9) and only then deallocates its memory (Line 10).

Then the secondary index is updated. All pointers to the logically deleted object are nullified (Line 11). The secondary index is also searched for *lastObj*'s entry so that it can be updated to store its new position in the grid index (Line 12). 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.

Algorithm 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 copied over ldObj
7      firstBckt.nO- -;                                          // decrement
8      if firstBckt.nO == 0 then                                 // is empty?
9          *sie.ldCell = firstBckt.nxt;
                                  // No more queries can enter firstBckt
           waitUntilNoReaders();
10         free(firstBckt);
11     Nullify all ld references in sie;
12     Lookup for lastObj's sie and update it;
13     unlockObj(lastObj); unlockCell(sie.ldCell);
14     return true;
15 else
16     unlockCell(sie.ldCell);
17     return false;
```

A failure to lock *lastObj* in Line 5 means that it is already locked by another thread. Instead of waiting until a lock can be obtained, the deleter unlocks the previously locked cell and returns with a failure indication (Lines 16–17). The unsuccessful return forces the update algorithm to restart its processing (Algorithm 1). This costly decision eliminates a potential deadlock. The deadlock would happen if the following two circumstances were to co-occur while running `delete`. First, another concurrent updater has to be updating the same *lastObj* object, implying that it is already locked in Line 2 of Algorithm 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 needs to obtain a lock on the same cell. The two updaters end up waiting for each other. Restarting one of the updaters solves the problem. Situations such as this are unlikely to occur, and our empirical study confirms that restarts are very rare.

Object insertion is relatively simple (Algorithm 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 the bucket is full, a new bucket is allocated, and the necessary pointers are updated so that the new bucket becomes the first (Line 4). The first free position at the end of the first bucket is determined (Line 5), and the new tuple is written (Line 6). The fields *cell*, *bckt*, and *idx* in *sie* are also updated accordingly (Lines 7–9). 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 and

Algorithm 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 copied over freePos
7  sie.cell = cell;
8  sie.bckt = firstBckt;
9  sie.idx = firstBckt.nO;
10 firstBckt.nO++;                                          // increment
11 unlockCell(cell);
```

k-nearest neighbor queries processing that satisfies freshness semantics.

4.3.1 Range Query

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 is used to distinguish between the two copies of the object. Algorithm 4 provides the details, which we proceed to describe.

Algorithm 4: `rangeQuery`(Rect q, int ts)

```
1  res = ∅;                             // container for storing the result[3]
2  cells = computeCoveredCells(q);
3  foreach cell ∈ cells do
4      objects = pCellScan(cell);                          // Algorithm 5
5      foreach obj ∈ objects do
6          if obj.tu > 0 then
7              res.addAndOverwrite(obj);
8          else if abs(obj.tu) > ts then
9              res.addIfNoSuch(obj);
10 Similar processing is performed for partially covered cells;
11 return res;
```

The algorithm takes two inputs: a two-dimensional rectangle specifying the query range (q) and an integer value specifying the query's timestamp (t_s). The algorithm returns the objects covered by the query range. For simplicity, Algorithm 4 shows only the processing of fully covered cells (computed in Line 2). Thus the extra check (whether an object is within the range) done for partially covered cells does not appear in the algorithm.

Objects from each overlapping cell are retrieved by scanning the cell (Line 4). The cell scanning is done in parallel with update processing, to be covered in more detail

shortly. 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; thus, it is taken into account (Line 7). If an object with the same *oid* already has been added, it can be replaced with the up-to-date position (add-and-overwrite).

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 (Line 8). This implies that the object was updated after the query started and so can be missed (see case i4 in Section 3.4). Second, the query has not yet seen the object's new (updated) position. The latter condition is realized via the container (Line 9): the tuple is added only if the result contains no object with the same *oid*. This as well as add-and-overwrite functionality can be efficiently supported using an associative container[2].

The crucial routine for parallel (w.r.t. updates) cell scanning is given in Algorithm 5. 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. However, the first bucket can have been deleted by the time the query actually increments its counter (Line 7). Therefore, a cell lock is acquired briefly (Lines 2 and 8). For the subsequent buckets, the counter can be accessed safely, as the next bucket cannot be freed before the first one (Lines 13–16). Atomic operations are used because the counters can be accessed by concurrent query threads[3].

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 10).

The following theorem states the correctness of the presented range query and update algorithms. It guarantees that range query results in PGrid always satisfy freshness semantics.

Theorem 1 *The* `rangeQuery` *algorithm (Algorithms 4–5), performed in the presence of concurrent* `update` *operations (Algorithms 1–3), returns results that satisfy freshness semantics.*

[2] We use `unordered_map` from the Standard C++ library.

[3] The cell locking in the scanning 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 contention or performance penalty due to this brief cell locking, we do not consider such techniques.

Algorithm 5: `pCellScan(Cell cell)`

```
1  objects = ∅;                          // container for storing the result
2  lockCell(cell);
3  if isEmpty(cell) then
4      unlockCell(cell);
5      return objects;
6  bckt = *cell;                         // cell refers to the 1st bucket
7  aInc(bckt.nR);                        // atomic increment
8  unlockCell(cell);
9  while bckt != null do
10     for idx = bckt.nO - 1 downto 0 do
11         obj = readObj(bckt.entries[idx]);
12         objects.add(obj);
13     if bckt.nxt != null then
14         aInc(bckt.nxt→nR);            // atomic increment
15     aDec(bckt.nR);                    // atomic decrement
16     bckt = bckt.nxt;
17 return objects;
```

We prove the theorem by examining all cases of Definition 2, where the most important part is to show that case i4 is avoided. The proof is given in Appendix B.

Note that Definition 2 does not require a specific object position ($pos^{\bullet}$ or pos°) to be returned by the range query. Therefore, in Line 7 of Algorithm 4, PGrid can also call add-if-no-such and still comply with freshness semantics: which position of an object is added depends on the (arbitrary) order the object versions are encountered by the query thread. This is useful in *k*NN query processing, described next.

4.3.2 kNN Query

Given a query point q at timestamp t_s, the *k*NN query has to retrieve k closest objects to q based on t_s according to freshness semantics (Definition 3). Algorithm 6 provides the details, which we proceed to describe.

Algorithm 6: `kNNQuery(Point q, int k, int ts)`

```
1  res = ∅;                              // container for storing the result
2  cd = +∞;                              // critical distance
3  cell = nextCell(q);                   // following [46]
4  repeat
5      objects = pCellScan(cell);        // Algorithm 5
6      foreach o ∈ objects do
7          if o.tu > 0 or abs(o.tu) > ts then
8              if dist(q, o) < cd then
9                  res.addIfNoSuch(o);
10                 if res.size() > k then
11                     remove furthest object from res;
12                     cd = max{ dist(q, o) | o ∈ res };
13     cell = nextCell(q);               // following [46]
14 until res.size() == k and cd <= dist(q, cell);
15 return res;
```

The underlying idea is to repeatedly read grid cells in best-first order until k objects are obtained and the distance from the query point to the next closest cell is no smaller than the critical distance (Line 14). The critical distance, denoted as *cd*, is the distance between q and the current k-th object (computed in Line 12). Also, given a cell *cell* and a query point q, $dist(q, cell)$ is the minimum possible distance between q and any object $o \in cell$. It can be computed in constant time.

To compute the next-best (closest) cell in Lines 3 and 13, we employ state-of-the-art techniques that partition the cells based on their position relative to the query point. Specifically, we follow Wu and Tan [46] (that improve on [29]) and divide the cells into levels and then divide the cells at each level into groups. The initial cell where the query point is located belongs to level 0. The neighboring cells around the initial cell form level 1, and the neighboring cells around the level l form level l+1. The cells at level l are divided into l+1 groups as illustrated in Figure 8. This enables us to access cells from closest to farthest by increasing level and group number.

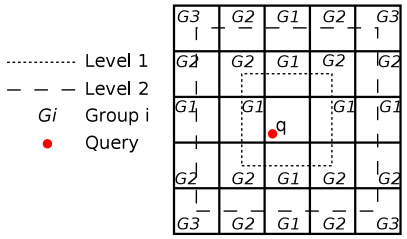


Fig. 8: Grid cell partitioning.

In the loop, the next cell is scanned in parallel with updates using the same algorithm as in range querying (Line 5). To be considered for the result, in addition to passing the freshness conditions (Line 7), each object has to be closer to q than the current critical distance (Line 8). If the result exceeds k objects, the farthest (k+1-st) object is removed, and the critical distance is updated (Lines 10–12).

As in range querying, PGrid uses an associative container to efficiently support add-if-no-such (*oid*) functionality. In kNN querying, PGrid needs to additionally maintain the k candidates (partially) sorted based on their minimum distance to q. Otherwise, *res* has to be scanned each time a new (old) candidate is added (deleted) and the critical distance updated. Therefore, PGrid employs a priority queue to enable constant-time access to the farthest candidate and logarithmic-time insertion/deletion. This functionality is implemented in container *res* (not shown).

Unlike for the range query algorithm, PGrid does not have a branch to add an object using add-and-overwrite. This is to avoid situations when the object's new position is no longer within the critical distance and a replacement has to be found. This implies that all candidates have to be maintained at all times (not only k). Moreover, whenever we need to compute *cd*, we have to sort *res*. The experimental study shows that this becomes very expensive. According to the following theorem, kNN query processing in PGrid satisfies freshness semantics.

Theorem 2 *The* `kNNQuery` *algorithm (Algorithms 5–6), performed in the presence of concurrent* `update` *operations (Algorithms 1–3), returns results that satisfy freshness semantics.*

We proove the theorem by examining all cases of Definition 3, where the most important part is to show that objects falling under case 1) (case 2)) are always included (excluded) from the result set O. The proof is given in Appendix C.

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 is achieved.

On 64-bit architectures, the reading and writing of 8-byte aligned quad-word values are guaranteed to be atomic. However, moving-object data occupies 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 during update [22] and lower accuracy [11] during retrieval. Another approach is to secure two 64-bit reads/writes using a lock, but such fine-grained locking incurs a significant overhead. In the following, we present two lock-free methods that are used for parallel reads and writes of moving object data in PGrid.

4.4.1 OLFIT

Optimistic lock-free index traversal (OLFIT) was designed as a cache-conscious concurrency control scheme for main-memory index structures on shared-memory multiprocessor platforms [7]. This approach maintains a latch and a version number for each object. Table 1 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 can occur 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.

Optimistic reads are especially favorable in multi-core architectures because they avoid the memory write required

Table 1: Read and write in OLFIT.

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

by latching. Thus, with latching, 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.

4.4.2 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 256-bit wide SIMD registers on the Ivy Bridge processor, one can add eight pairs of 32-bit integers simultaneously [18].

The instruction sets of such processors come with instructions for loading and storing data into SIMD registers. Thus, we can employ the SIMD technology in PGrid for parallel object data reads and writes. Our micro-benchmarks confirm that loading/storing a double quad-word (128-bit value) into/from SIMD registers (`xmm`) from/to a memory location aligned on a 16-byte boundary is atomic[4] [18]. Therefore, no explicit latching or locking are required.

To quantify the overheads of the above methods, we conduct a micro-benchmark, which leads us to employ the two lock-free methods for parallel object data reads and writes. The results and other findings are detailed in Appendix D.

5 Other Grid-Based Variants

We present three other parallel alternatives based on PGrid that support different query semantics as covered in Section 3. In all the variants, update messages are facilitated via the secondary index (hash table), and queries are processed via the grid directory. In the last subsection, we summarize all the indexes considered in the subsequent empirical study.

5.1 Serializable u-Grid

As a baseline approach to multi-threaded processing, we extend u-Grid [43] with the concurrency control scheme defined in Section 3.1 that ensures serializable execution and delivers timeslice query semantics. We term it *Serial*. It isolates updates and queries by using two-phase locking [5] and uses only cell locks. That is, unlike in PGrid, no object-level locking is performed by updaters.

Update threads obtain cell locks in exclusive mode. Before any object modification, each updater acquires a lock on the current object's cell. 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. In case the lock is obtained successfully, 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 aborted and restarted to avoid deadlocks.

Query threads obtain cell locks in shared mode. In range query, after all cells overlapping a query range are locked, the growing phase is completed, and the query enters the shrinking phase where it examines each cell in turn and releases locks one by one. The locks can be released safely before all the cells are inspected because range queries are read-only and never abort.

In the *k*NN query, Serial partitions the cells only into levels (no groups, see Section 4.3.2). The growing phase incrementally locks an enlarged search region level by level until the locked cells cover at least k objects. In this phase, only counters of buckets are read (no actual cell scanning). In the shrinking phase, the query scans each cell, maintaining the k closest candidates and releasing locks one by one. After the completion of this phase, if the minimum distance between the query point and the next level cell is less than the critical distance, the *k*NN query is repeated, but this time locking one level more. We found that this approach works best. Finer cell partitioning (into groups) makes the expensive repetitions more likely, while holding cell locks till the end (strict two-phase locking) does not scale at all with multi-threading (although it eliminates repetitions).

We expect exclusive locks to be held for very short durations of time because the fixed and static grid design does not suffer from long-running structural modifications (no grid refinement or re-balancing). At most two cells are locked during an update. Also, shared locks are not held til the end of the query (non-strict two-phase locking). This implies that Serial should perform well with a small number of threads or when updates and queries write and read objects in different cells, i.e., when hot spots are unlikely.

5.2 Serializable PGrid

An alternative way to achieve the same serializable timeslice semantics as implemented by Serial is to augment PGrid to support MVCC. In contrast to the traditional MVCC, where multiple versions of each data record are maintained, only

[4] Note that we achieved this atomicity only with 128-bit values. That is, our experiments break once we use 256-bit values and `ymm` registers.

two versions are necessary due to the assumption $t_e - t_s < T_o$. As discussed in Section 3.2, such processing runs under snapshot isolation and results in serializable executions with the given update and query operations. We thus term this extension *SerialPGrid*.

SerialPGrid reuses most of the code in PGrid. The same locking mechanism is employed (using one-byte latches) to isolate updates, while not blocking queries. Object data is also written using OLFIT/SIMD techniques. The update procedure reuses the deletion and insertion given in Algorithms 2 and 3. The update algorithm is modified so that when an object is inserted for the first time, two tuples are allocated in a bucket. As discussed next, the algorithm tries to reuse the allocated tuples by continuously storing current and previous object positions in a round robin manner. The range and *k*NN query algorithms are modified to take into account only tuples with timestamps smaller than the query timestamp.

Algorithm 7 shows the pseudo-code of the modified update algorithm. The first lines (Lines 1–5) are the same as in PGrid. Then, the algorithm checks whether the previous logically deleted copy of an object can be reused. This is done by comparing the new cell with the cell that the logically deleted object belongs to (Line 6). If the cells are the same, the previous LD tuple is reused by writing the new object data over it (Line 8). The corresponding references in the secondary index entry are swapped (Lines 9–11), and the current position is now marked as logically deleted (Line 12). This way, the roles of the two tuples are switched: the logically deleted one becomes the current, and the current becomes the logically deleted one.

In case the object's new position belongs to a different cell than its logically deleted position belongs to, the tuple cannot be reused and thus is physically deleted (Line 14). For the same reasons as in PGrid, the update is restarted if the deletion fails. Otherwise, the current position becomes logically deleted (Lines 17–19, and 21), and the new position is inserted into the new cell (Line 20).

Algorithm 8 shows the lines of Algorithm 4 that need to be modified to support serializable range query semantics. Specifically, now the query reports `obj` in the following two cases. Object `obj` is added to the result (i) if it contains the up-to-date position and the position was updated before the query start (Lines 6–7), and (ii) if it contains the previous object position and the up-to-date position is recorded after the query start (Lines 8–9). Similarly, in the *k*NN query, Line 7 of Algorithm 6 is modified accordingly. Note that SerialPGrid does not require an associative container, as only one version of the object can be added to the result.

We expect SerialPGrid to achieve the same level of concurrency as PGrid, as updates and queries never block each other. However, the performance not related to contention suffers. Each updater now does extra work in maintaining the previous object position for each moving object rather than just for non-locally updated objects. Consequently, queries must inspect twice as many tuples on average. As a result, memory consumption doubles and becomes the same as in the full snapshot-based approaches (e.g., TwinGrid [42]).

Algorithm 7: `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 sie.ldCell == newCell then
       /* Can reuse LD object, so write to it */
7      ldObj = getObj(sie.ldBckt, sie.ldIdx);          // object tuple
8      writeObj(new, ldObj);                 // new copied over LD obj
       /* Swap references */
9      swap(sie.cell, sie.ldCell);
10     swap(sie.bckt, sie.ldBckt);
11     swap(sie.idx, sie.ldIdx);
12     obj.tu = -new.tu;                     // mark as logically deleted
13 else
       /* Can't reuse LD object, so delete it */
14     if !delete(sie) then                          // physical deletion
15         unlockObj(new.oid);
16         go to 2 ;                                          // try again
       /* Up-to-date position becomes LD */
17     sie.ldCell = sie.cell;
18     sie.ldBckt = sie.bckt;
19     sie.ldIdx = sie.idx;
       /* And new up-to-date is inserted */
20     insert(new, newCell, sie);                // physical insertion
21     obj.tu = -new.tu;                     // mark as logically deleted
22 unlockObj(new.oid);
```

Algorithm 8: Modified lines in Algorithm 4

```
6  if obj.tu > 0 and obj.tu < ts then
7      res.add(obj);
8  else if obj.tu < 0 and abs(obj.tu) > ts then
9      res.add(obj);
```

5.3 Time-Partitioning Grid (TP-Grid)

We proceed to consider a time-based approach to isolating update and query operations. Specifically, we partition the processing into two phases: an update phase (with duration P_u) and a query phase (with duration P_q). We call this approach *TP-Grid*.

Figure 9 illustrates how processing is done by switching between the two phases. Once the desired number of threads is spawned, progress is made by switching threads between different processing modes. Synchronization barriers are used to ensure that no thread starts in the next phase before all other threads have suspended their processing in the current phase.

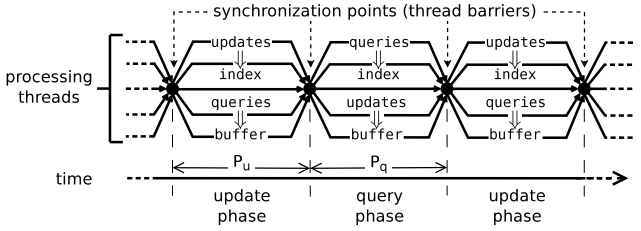


Fig. 9: Two-phase switching in TP-Grid.

During the update phase, only updates are applied to the index structure. This includes newly incoming updates and updates buffered in the previous phase. All incoming queries are buffered. During the query phase, only read-only queries are processed, including previously buffered queries and incoming queries. All incoming updates are buffered for processing in the subsequent update phase. This switching continues until the entire workload is processed.

This approach results in the stale-timeslice query semantics described in Section 3.3, and it suffers from query staleness and latency. The query staleness depends on the value chosen for duration P_q. Since all updates are buffered during the query phase, parameter P_q defines the maximum query staleness in TP-Grid. This is similar to the discussed snapshotting frequency (F_s).

Similarly, the query latency depends on the value chosen for duration P_u. The processing of queries that arrive during the update phase is delayed until the next phase. Thus, the maximum query latency is defined by P_u. Note that the processing of updates is delayed similarly. However, since an update does not create any visible output, it can be perceived as having been processed without any delay, e.g., the system can respond with a confirmation message as soon as the update is received.

Given that update and query rates are not uniformly distributed across time [40], it might be very challenging to choose the durations for each phase optimally. In the following, we list several possible policies:

1. Based on time, e.g., $P_u = 1$ sec, $P_q = 0.5$ sec.
2. Based on the number of operations processed, e.g., $P_u =$ 100 updates, $P_q = 10$ queries.
3. Hybrid (1), e.g., $P_u = 100$ updates or at most 1 sec, $P_q =$ 10 queries or at most 0.5 sec.
4. Hybrid (2), e.g., $P_u = 100$ updates or idle at most 0.5 sec, $P_q = 10$ queries or idle at most 0.1 sec.
5. Based on quality of service, e.g., $P_u =$ to the maximum query latency allowed, $P_q =$ to the maximum query staleness allowed.

The last policy adapts to the incoming workload, and we use it in our empirical study (see Section 6.4). P_u is set as a time-based parameter equal to the maximum query latency. The maximum query latency might not be guaranteed completely as it takes some time to process all the buffered queries at the start of the query phase. P_q, on the other hand, should not be set to a specific value, but should be based on the query staleness. More specifically, the query phase should stop when the number of updates accumulated in the buffer in relation to the total number of tracked objects reaches the query staleness.

On the negative side, TP-Grid provides stale and delayed query results and relies on two parameters that are potentially hard to set. On the positive side, we expect that TP-Grid can achieve very high throughput. It achieves the same level of parallelism as PGrid and SerialPGrid, while the performance not related to contention is at the level of Serial. Updates do not need to maintain extra object copies, while queries do not need to scan them. Also, during each phase, all threads (and other CPU resources) are dedicated to the type of operation performed in the phase. During the update phase, it is more likely that the most frequently updated objects can be found in the CPU data caches (and are not flushed by parallel queries). During the query phase, the cached objects are not invalided by parallel updates. Instruction caches are also used more effectively because TP-Grid issues fewer instructions per phase.

5.4 Summary of Approaches

The list below summarizes the eight grid-based implementations that we study experimentally in the subsequent section.

`u-Grid:` a single-threaded, update-efficient index that was shown to outperform update-optimized R-tree index variant [43].

`Serial:` a multi-threaded u-Grid variant supporting serializable semantics (described in Section 5.1).

`TwinGrid:` a multi-threaded, snapshot-based index that was shown [42] to outperform the previous state-of-the-art snapshot-based approach called MOVIES [11].

`TP-Grid:` a multi-threaded, time-partitioning index, where processing is done by switching between update and query phases (described in Section 5.3).

`PGrid`$^{\texttt{simd}}$`:` a PGrid variant supporting freshness semantics and exploiting SIMD technology for reads and writes [44].

`PGrid`$^{\texttt{olfit}}$`:` a PGrid variant supporting freshness semantics and exploiting the OLFIT [7] approach for reads and writes [44].

`SerialPGrid`$^{\texttt{simd}}$`:` a PGrid variant supporting serializable semantics and exploiting SIMD technology for reads and writes (described in Section 5.2).

`SerialPGrid`$^{\texttt{olfit}}$`:` a PGrid variant supporting serializable semantics and exploiting the OLFIT [7] approach for reads and writes (described in Section 5.2).

Enabling a higher degree of parallelism involves different indexing trade-offs. Figure 10 categorizes the indexes

in terms of query semantics and query-result freshness. The figure shows that two techniques achieve better throughput at the expense of returning results that are stale or at the expense of returning results late. TwinGrid (immediately) processes queries against a (slightly) stale copy of the data, while TP-Grid (slightly) delays the processing of queries that arrive in the index's update phase and (immediately) processes queries that arrive in the query phase against a (slightly) stale copy of the data. Serializable executions in u-Grid, Serial, and SerialPGrid guarantee up-to-date timeslice semantics. PGrid implements freshness semantics that guarantee as fresh query results as possible.

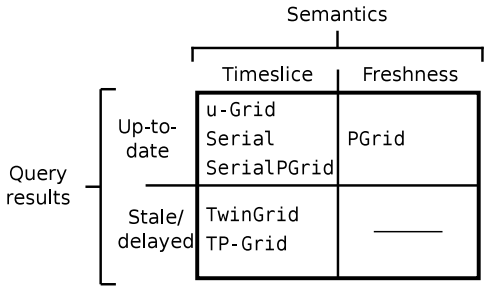


Fig. 10: Index classification.

6 Performance Study

We investigate the performance of the four PGrid variants and the additional four indexing techniques outlined in Section 5.4. Before reporting the findings, we describe the experimental setting, explore how to set the snapshotting period for TwinGrid [42] so as to achieve a fair comparison, and consider the setting of the durations of the update and query phases in TP-Grid.

6.1 Experimental Setup

We study the performance on diverse multi-core platforms: a 12-core AMD Opteron 6176 (Magny-Cours), a dual 6-core Intel X5650 (Westmere) with 12 hardware threads, a dual 8-core Intel E5-2670 (Sandy Bridge) with 32 hardware threads, and a quad-core Intel Core i7-3770 (Ivy Bridge) with 8 hardware threads. All machines have enough main memory to store both the entire workload and the populated index. More details are given in Appendix E.

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[5] [11] that is based on Brinkhoff's moving-object generator [6]. MOTO follows a network-based object placement approach, where objects are placed and navigate (to a random destination) in a given road network. Table 2 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. Since kNN queries involve range scanning, the default workloads consist of range queries only.

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

Table 2: Workload configuration.

Parameter	Values
objects, $\times 10^6$	5, **10**, 20, 40, 80
updates, $\times 10^6$	300
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
update/query ratio	250:1, ... **1000:1**, ... 16000:1
time between updates (T_o), s s	**10**, 20, 40, 80, 160
range query size, km^2	0.25, 1, **4**, 16
k in nearest neighbor query, $\times 10^3$	0.5, 1, **2**, 4, 8, 16, 32

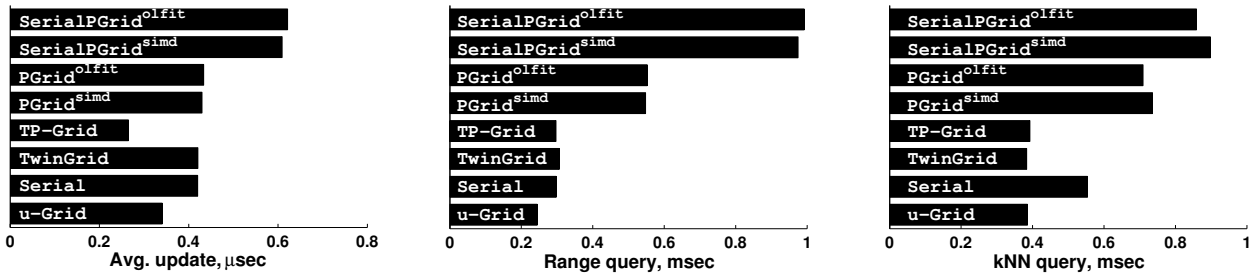
6.2 Single-Threaded Performance

We start by providing an overview of the performance unrelated to contention, where each index processes the default workload using just one thread. Figure 11 shows the average CPU time of individual update, range, and kNN query operations spent in each index.

In update processing (Figure 11a), TP-Grid spends the least time and outperforms even the lockless u-Grid. This is because updates are not intermixed with queries during the update phase, causing less data and instruction cache misses in TP-Grid. Serial and TwinGrid performs slightly worse than u-Grid. This implies that the one-byte latching overhead is minor (circa 15%). Overhead due to maintenance of previous object versions for non-locally updated objects in PGrid is also negligible. However, the maintenance of previous object versions for all moving objects in SerialPGrid requires almost double the CPU time when compared to u-Grid.

In range query processing (Figure 11b), three major categories can be distinguished. First, Serial, TwinGrid, and TP-Grid perform very similar as u-Grid. This is as expected in the zero contention case. Second, the need to inspect extra copies during cell scanning causes the two PGrid variants to perform twice as bad as u-Grid. Third, for the same reason, the two SerialPGrid variants suffer from four times as bad performance as u-Grid.

In kNN query processing (Figure 11c), we choose the default k value (2000) so that a kNN query completes in a



(a) Average CPU time per update (b) Average CPU time per range query (c) Average CPU time per kNN query

Fig. 11: Single-threaded performance.

time similar to that of the default range query. Given that the kNN query is much more expensive (computing $dist(q,o)$ for each object, maintaining cd, etc), the actually scanned range is much smaller than in the above range query. Therefore, the overhead due to reading multiple object copies becomes less significant. As a result, SerialPGrid is not outperformed by the same margin as in range querying. Serial performs clearly worse due to a less granular cell partitioning (more cells are scanned than necessary) and a penalty of occasionally repeating the scan.

There is no clear winner among the SIMDized and OLFITized PGrid variants. Unlike in the previous micro-benchmark (see Table 4), we do not observe any significant savings. This implies that a bigger fraction of cycles is consumed elsewhere. Since the OLFIT variant does not depend on atomic 128-bit loads and stores, use of it makes PGrid the most architecture-friendly (portable).

In all indexes, the processing of an update completes within one microsecond, while the processing of a range query completes within one millisecond. This is a difference of three orders of magnitude. Therefore, if updates are blocked by queries during parallel processing, system throughput can decrease sharply.

6.3 Setting the Snapshotting Frequency in TwinGrid

Serializable and freshness semantics enable indexes with zero query staleness, whereas stale-timeslice semantics suffer from outdated and delayed query results (see Section 3). In two-snapshot isolation based approaches like TwinGrid, the query staleness 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 ignores all updates that occur during the time between the creation of consecutive snapshots (the cloning period, T_{cp}, in TwinGrid). In the following, we explore experimentally how to tune TwinGrid so that it returns sufficiently fresh query results to achieve a fair performance comparison.

To maintain the desired query staleness in TwinGrid, we perform snapshotting based on the number of processed update messages. For example, if we want to guarantee that the query staleness never exceeds 1% under the default workload, we perform snapshotting every time 100K updates (1% of 10M) are being processed.

Figure 12 depicts effects of maintaining different query staleness values (the x-axis) in TwinGrid. The effects on throughput are shown on the y-axis, while the bars break the CPU cycles into cycles spent on actual computation (processing) and cycles wasted on cloning (snapshotting). The experiments confirm that keeping the query staleness below 1% is indeed feasible, but also very expensive: circa 60% of the CPU time is spent on snapshotting. This is a huge waste of computation resources, as the CPU needlessly copies the 99% of the data that is unchanged. For example, relaxing the query staleness from 1% to 8% results in a two-fold throughput improvement.

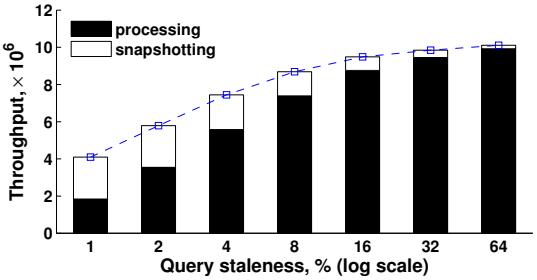


Fig. 12: Effects of varying the query staleness in TwinGrid on Ivy Bridge.

It is not trivial to choose the desired query staleness in TwinGrid for the sake of fair comparison with the other index structures, as they report up-to-date query results. Nevertheless, in the following experiments we fix TwinGrid's query staleness to 2% so that a larger fraction of CPU resources are used on actual processing (choosing a lower query staleness would be fairer w.r.t. other indexes, though).

6.4 Setting the Update/Query Phase Durations in TP-Grid

Similarly to the snapshotting frequency in TwinGrid, we need to set how frequently to switch between the update and query phases in TP-Grid. If we choose values for P_u/P_q based on the number of updates/queries processed, we can control the quality of service parameters (policy 5 in Section 5.3) as follows. For example, to maintain the query staleness below 1% under the default workload, we set P_q = 100K updates (1% of 10M). When the number of buffered updates reaches 100K, TP-Grid switches to the update phase. We set P_u also to 100K, as all of the buffered updates have to be processed. This is the minimum query latency TP-Grid has to pay.

Figure 13 shows the results. Different values for query staleness are depicted on the x-axis and its effect on throughput is projected on the left y-axis. The bars split the CPU time spent in each phase. The number of phase switches per second are projected on the right y-axis (grey bars).

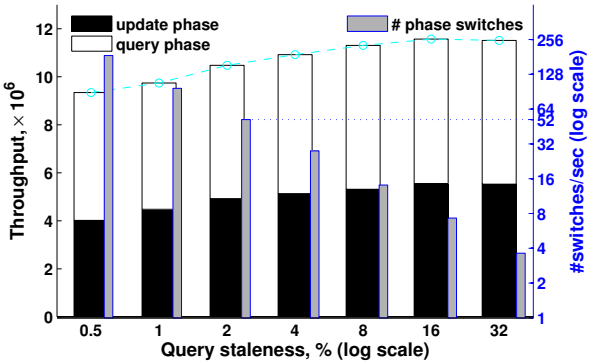


Fig. 13: Effects of varying the query staleness in TP-Grid on Ivy Bridge.

The cost of frequent phase switching is relatively minor. A throughput of more than 9M update/query messages per second is sustained while keeping the query staleness as low as 0.5%. Relaxing the query staleness up to 32% (64 times) improves the throughput only by some 20%. This is despite the fact that the number of phase switches per second decreases significantly: from 186 to less than 4. The update and query phases are of a similar duration: on average 46% of the CPU time go to the update phase, and 54% go to the query phase.

As discussed in Section 5.3, TP-Grid also suffers from operational latency. Updates and queries might not be processed as soon as they are received. The latency depends on the phase change frequency. For example, a 2% query staleness yields 52 phase switches per second (Figure 13). This implies an average phase duration of $\frac{1}{52}$ of a second. Since update and query phases are of similar duration (46% vs. 54%), the expected maximum latency is circa 19 msec, respectively.

For a fair comparison, we use the same query staleness (2%) in TP-Grid as in TwinGrid in the following experiments. This choice is also justified by the following observation. In TwinGrid, to make a 10M-object snapshot takes 20–27 msec on our experimental platforms. If an update or query arrives during this snapshotting, it suffers the latency of the snapshotting duration. This is a very similar duration to the expected maximum latency discussed above when TP-Grid is configured with a 2% query staleness.

Note that to choose P_u and P_q values based on the number of processed operations is straightforward in our setting because we know the workload parameters. This serves the purpose of measuring the peak performance of the indexes very well. In real-world applications, the values are likely to be based on time or hybrid options (see Section 5.3).

6.5 Experimental Findings

6.5.1 Varying the Grid Cell Size and the Bucket Size

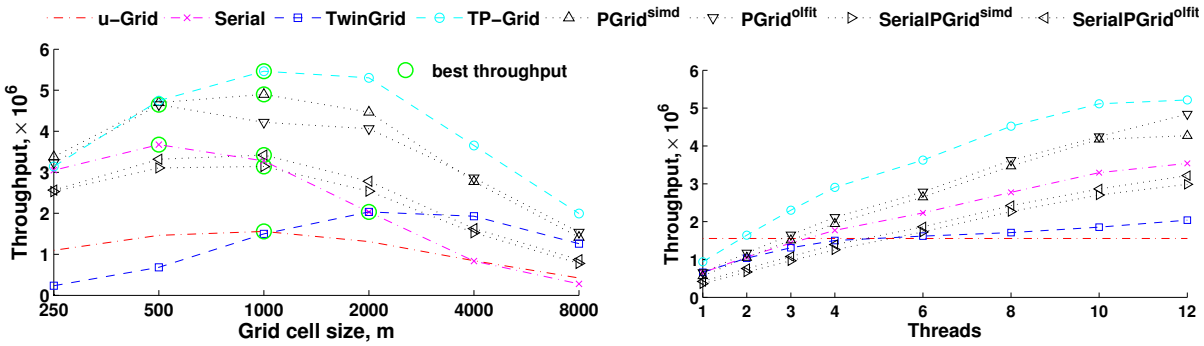
Two important parameters affect the performance of all the indexes: the grid cell size and the bucket size. To set the values of these parameters according to our workloads, we deploy all indexes on each processor while trying different configurations. For example, we vary the grid cell size from 250 to 8,000 m (side length). The results from the all processors are shown in Figure 14. Grid cell sizes are on the x-axis, and 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 operate on separate index structures. In contrast, Serial suffers from conflicts due to extensive cell locking. It thus favors cells that are four times smaller. The PGrid variants and TP-Grid fall between the two and achieve the best throughput with TP-Grid on top. The optimal grid cell sizes for each index are chosen as indicated by their best performance (circles in Figure 14).

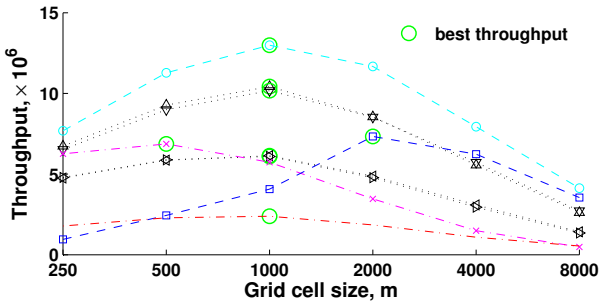
The bucket size has a relatively smaller impact on performance, but it was tuned in a similar manner. We found that the optimal bucket size is around 1024 bytes on all platforms (not shown).

6.5.2 Multi-Threaded Scalability

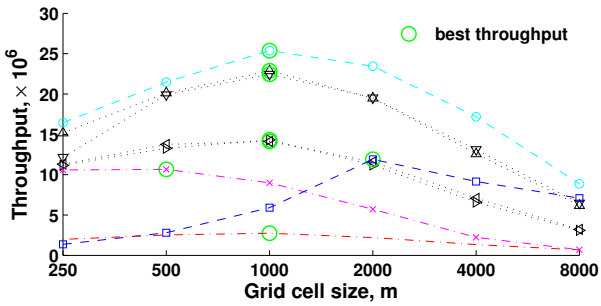
Figure 15 shows how the indexes scale with increasing numbers of hardware threads on the different platforms. The num-



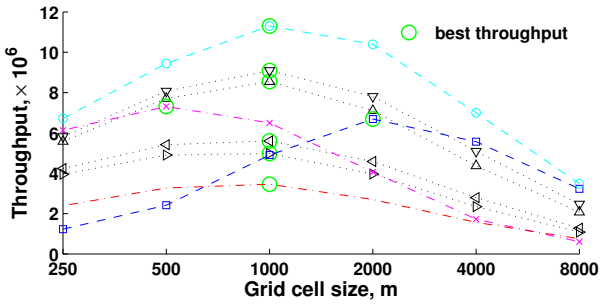
(a) 12-core Magny-Cours



(b) 12-core Westmere



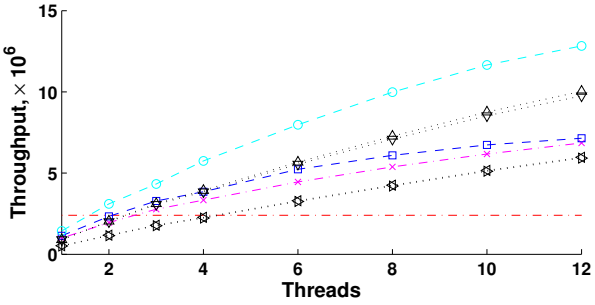
(c) 16-core Sandy Bridge



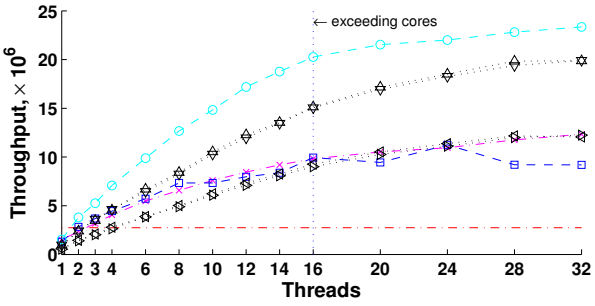
(d) 4-core Ivy Bridge

Fig. 14: Optimal grid cell size.

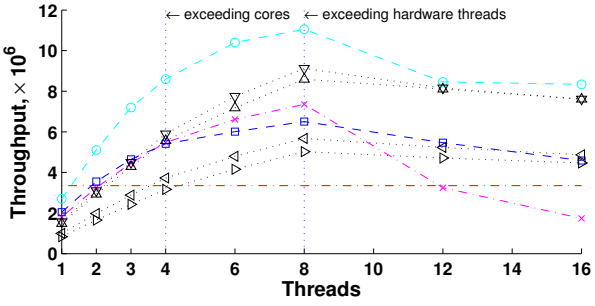
(a) 12-core Magny-Cours



(b) 12-core Westmere



(c) 16-core Sandy Bridge



(d) 4-core Ivy Bridge

Fig. 15: Multi-threaded scalability.

ber 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 read and write operations becomes. Therefore, with each additional thread, the performance gain tends to decrease. The studied multi-threaded indexes are subjected to different amounts of interference, but all outperform the single-threaded u-Grid approach with 2–4 threads. This implies that the overhead due to added thread-level synchronization pays off already on a

quad-core platform. Another common trend is that one must be careful not to exceed the available hardware threads on a given platform, as performance can quickly become worse than that of the single-threaded u-Grid. We intentionally do this on Ivy Bridge machine (Figure 15d), but the same trend holds on all platforms. The decreased performance occurs because as soon as the hardware threads are exhausted, a latch-holding thread has the potential to be "parked" by the scheduler, forcing other threads to busy-wait without progress. This adds many "busy" waiting times during the processing.

Serial is quite competitive with a small number of threads. Thus, for processing with up to 4 threads, Serial is a reasonable choice. However, with more threads, the gap increases, and Serial falls behind. This is due to increased interference between update-update and update-query operations. Serial's sensitivity to the thread-level contention is especially visible once the hardware threads are exhausted, as its performance decreases the sharpest (Figure 15d).

TwinGrid suffers from relatively rare update-update interference. However, due to frequent cloning, TwinGrid often has to suspend and resume worker-threads so that a consistent data snapshot can be made. During cloning, all CPU caches are flushed, which prevents processing threads from enjoying warm caches. Also, the more threads that are running, the more costly it is to synchronize them. This frequent and continuous interruption of the actual processing limits TwinGrid's throughput on all platforms.

The PGrid variants are also prone to the relatively rare contention between update operations, but updates do not interfere with queries. Therefore, the performance of the PGrid variants increases with each additional thread. For SerialPGrid, however, updating and querying is more expensive: each update has to maintain previous object versions, and each query has to inspect twice as many object tuples on average. As a result, the overall throughput in SerialPGrid is low.

PGrid creates extra object copies only during non-local updates, which also causes less work for querying. Consequently, PGrid scales near linearly on all platforms and utilizes all hardware threads much more efficiently. As in the single-threaded experiment (Section 6.2), there is no significant difference among the SIMDized and OLFITized variants.

The time-partitioning approach in TP-Grid scales best with an increasing number of threads. The synchronization cost due to frequent phase switching is not visible even when running all 32 hardware threads on the Sandy Bridge machine. TP-Grid reports the best throughput on all the platforms. The biggest performance gap compared to other indexes is achieved on the Ivy Bridge machine, where relatively small number of threads operate at the highest frequency (3.4 GHz) and thus can synchronize between the phases with the least overhead. TP-Grid does not guarantee fresh query results (the query staleness is 2%), though. Also, for the same reasons as Serial, its performance degrades sharply once the available hardware threads are exhausted.

In the following experiments, the number of threads is fixed at the number of available hardware threads. Also, since the same performance trends were observed on all of the processors, we show the results on the best performing platform—the 16-core Intel E5-2670 (Sandy Bridge) with a total of 32 hardware threads. The SIMDized and OLFITized PGrid variants perform very similarly, and we therefore exclude the OLFIT variants from the following graphs. We also exclude the single threaded variant u-Grid, as it is not competitive with the multi-threaded indexes.

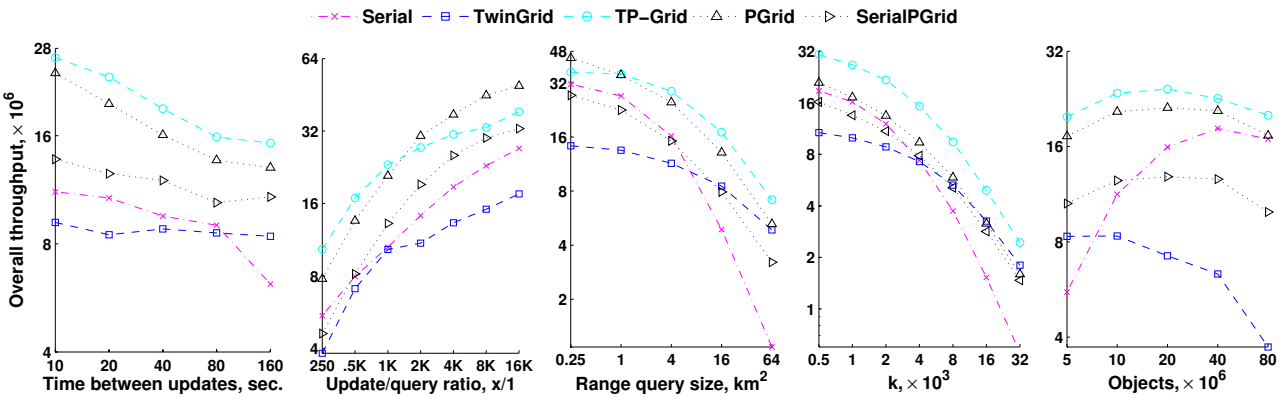
6.5.3 Varying T_o

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, TwinGrid, and TP-Grid require extra locking while an object is being deleted from its old cell and inserted into its new cell. PGrid creates a logically deleted copy of the object for each non-local update, and it deletes it at the next update. SerialPGrid also has to physically delete the previous object position once it belongs to a new cell and cannot be reused.

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's new position from its previous position becomes. Figure 16a shows the results. All indexes are affected negatively, but with different impact. Remember that the indexes are configured with different grid cell sizes for optimal performance (Figure 14), so they are subjected to different amounts of local updates. In Figure 17, we measure how the percentage of local updates varies with different grid cell size and T_o values. For instance, TwinGrid, configured with the largest cells ($\geq$ 2,000 m), has at least 60% local updates even when $T_o = 160$ s. Therefore, its throughput is affected the least. On the opposite side, Serial is configured with the smallest cells (500 m) and is affected the most—its performance drops sharply. The PGrid variants and TP-Grid are configured with in-between cell size and perform the best, with TP-Grid on top.

6.5.4 Varying the Update/Query ratio

Figure 16b depicts the results when the update/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



(a) Varying T_o (b) Varying update/query r. (c) Varying range query (d) Varying kNN query (e) Varying #objects

Fig. 16: `Serial` vs. `TwinGrid` vs. `TP-Grid` vs. `PGrid` vs. `SerialPGrid` on a 16-core Sandy Bridge machine.

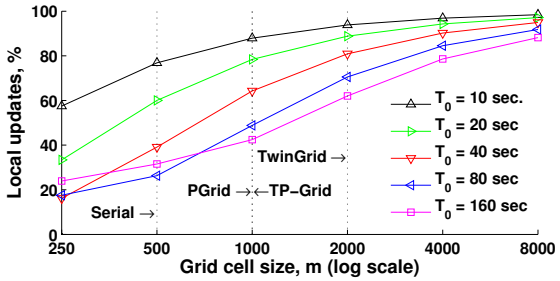


Fig. 17: Varying grid cell size and T_o.

the workloads decreases. The increasing update/query ratio causes TwinGrid and TP-Grid to switch between the phases more frequently, as the desired (2%) query staleness has to be maintained. Therefore, their performance gain is lower. Also, Serial benefits more from the increasing update/query ratio because infrequent queries cause less interference with updates. In this experiment, the PGrid therefore is more robust and eventually achieves the best performance.

6.5.5 Varying the Range Query Size

The impact of varying the range query size is shown in Figure 16c. 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 the least and should be considered under workloads with very large queries (more than 64 km^2). The performance of PGrid and SerialPGrid decreases in a similar fashion with PGrid on top. With query sizes different from the default workload, the fixed durations for update and query phases in TP-Grid become suboptimal and requires re-calibration. As a result, the throughput is just slightly better than that of PGrid with much smaller or larger queries.

6.5.6 Varying k in kNN Query

In this experiment, all range queries are substituted with kNN queries. Figure 16d shows the impact of varying k—the number of nearest neighbors required. Similar to range querying, increasing k decreases the performance of all indexes, as a larger range has to be scanned, which also increases the interference between update and query threads in Serial. The main difference is that PGrid and SerialPGrid are very competitive. This is because the overhead due to reading multiple object copies is less significant in kNN querying. Serial remains competitive until relatively large k (4000), but eventually drops behind.

6.5.7 Varying the Number of Objects

In this experiment, we check how the index variants scale with an increasing number of indexed objects. Since the monitored region does not change (the road network of Germany), increasing the number of objects increases object density. This implies that a query needs to inspect more objects per square unit on average. Therefore, in this experiment, we halve the range query size when doubling the number of objects. (The effects of increasing query costs are already explained by the previous experiment in Figure 16c.)

As the number of objects increases, the updates and queries operate on an increasingly large data set, which eventually adversely affects performance for all indexes (Figure 16e). TwinGrid suffers the most because with increasing data size, it needs to copy more data and the snapshotting phase becomes longer. In the case of Serial, the main bottleneck is the interference between update and query operations. Thus, we observe that Serial benefits much more from halving the query range than it suffers from doubling the number of objects. The throughput of Serial increases up to a load of 40M objects, where it becomes very competitive. Therefore, as with low numbers of threads, if interfer-

ence between updates and queries is unlikely, Serial can be a good choice. Otherwise, PGrid performs better and should be considered if fresh query results are required. The performance of SerialPGrid follows the same trend as that of PGrid, but is lower due to extra computations. At the cost of a 2% query staleness, TP-Grid performs the best and outperforms PGrid by up to 18%.

6.6 Summary

All the indexes we study exhibit increased performance with increasing numbers of threads/cores. This is mainly due to the careful implementation of the update-efficient, static, and uniform grid. However, the gain from each additional thread varies significantly across indexes because they support different query semantics and have varying overhead unrelated to contention.

The best throughput is achieved by the time partitioning approach. Under the default workload, more than 23M messages per second are processed on the 16-core platform. However, TP-Grid offers stale-timeslice semantics, and its query results are generally not up-to-date. On top of that, the query processing can be delayed by a few orders of magnitude longer than the time taken by the query processing itself. This might not be acceptable in application scenarios where there is no human in the loop. Another drawback is the difficult-to-set parameters. The durations of the update and query phases may be difficult to set. Since no fixed values may offer robust performance when the workloads vary across time, adaptive techniques are called for. However, we find that the TP-Grid approach is superior to the snapshot-based approaches such as TwinGrid. Therefore, we recommend TP-Grid over TwinGrid in applications where stale-timeslice semantics are applicable.

PGrid achieves the second best throughput. Under the default workload, some 20M messages per second are processed on the 16-core platform. With the other workloads, PGrid exhibits up to 25% lower throughput when compared to TP-Grid. Nevertheless, it provides the most robust performance and guarantees as fresh query results as possible. All queries are executed as soon as they are received and operate on the most up-to-date state of the data. Moreover, it does not suffer from the difficult-to-set parameters. Therefore, we believe this approach is applicable in all the target applications.

The extension of PGrid with time-slice query semantics, SerialPGrid, performs at 50% of PGrid. This is due to the versioning overhead for each moving object. In fact, it performs very similar to the serializable approach Serial (except in cases with very high update/query interference, e.g., with very large range queries). Since Serial is straightforward to implement, we recommend it over SerialPGrid.

We recommend using the OLFIT technique over the SIMD technique for two main reasons. First, the SIMD savings are overwhelmed by other operations, and we do not observe any performance gain. Second, the OLFIT technique is architecture independent and thus more portable. Similarly, we recommend using 1-byte latches (or spin locks) using the atomic CAS instruction.

7 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 [31]. We also focus on the indexing of current object positions.

Existing concurrency control (CC) protocols for multi-dimensional indexes [8,23,24,30,35,37] 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 [17,37] and then to allow concurrent insertions and deletions in the search region. Several phantom-protecting CC schemes for multi-dimensional indexes have been proposed [8, 24]. 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. [17] compare six main-memory R-tree variants, including their concurrent performance. They find that conventional latch-based CC using the link technique [24] (originally proposed for the B-tree [27]) does not scale: an 8-CPU platform is exhausted with just 6 threads under a simple update-only workload. Other benchmarks also suggest that lock-based approaches are inefficient [9]. Further, a study [17] shows that lock-based R-tree variants are outperformed by counterparts that utilize optimistic latch-free index traversal (OLFIT) [7]. We make use of the OLFIT technique and describe it in detail in Section 4.4.1. Since R-trees inherently suffer from CPU-intensive update algorithms (splitting/merging of nodes, propagating minimum bounding rectangle changes, etc.), we choose to use a grid-based index. While we are aware that adaptive approaches like R-trees can deal better with highly skewed workloads,

the data skew in the target applications is limited by the underlying road-network. Previously proposed uniform grid-based approaches have proven to be superior for update-intensive moving object applications [21, 42, 43, 46, 47].

One way to avoid interference between parallel queries and updates is to let queries and updates operate on different data structures [11, 42]. The query structure is read-only and so does not require locking. Single-object updates can be parallelized using atomic operations[6], by partitioning the data into disjoint sets and allowing only one update thread per set (shared-nothing [38]) or by using a simple concurrency control scheme [42]. 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, resulting in a substantial waste of CPU cycles.

Based on a *copy-on-write* mechanism, a granular rebuilding technique has been proposed [35]. 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 relatively modest update rates.

Support for location-based applications has recently been studied in a cloud computing setting [20, 32]. There, the processing is parallelized and distributed across a multi-node cluster with the main goal to minimize inter-node communication. This is achieved by shared-nothing [32] or update shedding [20] approaches. However, such systems often suffer from poor per-node efficiency, e.g., data-intensive tasks in such systems utilize only 5–10% of each individual node's capabilities [2]. Therefore, if well optimized, a single (multi-socket) platform can replace a cluster with several nodes. Our work focuses on efficient parallel processing within a single node.

This paper substantially extends a previously published conference paper [44]. In that paper, we define freshness semantics that ensure fresh query results as well as enable a high-degree of parallelism, and we propose PGrid that supports the defined semantics. This paper re-investigates the possible query semantics in the context of conventional transaction processing within database management systems (Section 3). In addition to range queries, it investigates what guarantees are provided for k nearest neighbor query results when moving-object updates and queries are processed in parallel. Next, this paper in addition considers two alternative approaches. In Section 5.2, the first approach extends PGrid to support serializable semantics via snapshot isolation (described in Section 3.2). This is done by keeping track of previous object positions for all indexed objects. In Section 5.3, the second approach uses time for the partitioning of updates from queries, thus having separate phases for updates and queries. In this way, performance is increased at the cost of higher latency and outdated query results (stale-timeslice semantics, described in Section 3.3). Lastly, the entire empirical study is carried out on newer and more parallel multi-core platforms (Section 6). This includes an analysis of how to set the snapshotting frequency in TwinGrid (Section 6.3) and the phase switching frequency in TP-Grid (Section 6.4) optimally. The study takes into account all the eight indexing techniques considered in the paper and offers new insights into the scalability of the techniques with continuously increasing parallelism.

[6] For instance, 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.

8 Conclusions

Increased on-chip parallelism is a key means of improving processor performance. This development calls for software techniques that are capable of scaling near-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 investigate different levels of update and query isolation, including fully serializable isolation, snapshot-based isolation, and isolation via the time-partitioning approach, where only updates or only queries are processed during a time phase. The different levels of isolation result in different query semantics and enable different degrees of parallelism. Consequently, we define freshness query semantics that enable a high-degree of parallelism and ensure the most up-to-date query results.

All investigated semantics are implemented using a static grid-based index structure and enacted with different workloads on four diverse multi-core platforms. Key advantages of our main proposal, called PGrid, include up-to-date query results and the ability to make efficient use of thread-level parallelism. Compared to the snapshot-based approach (TwinGrid), PGrid creates multiple copies of data on demand and at object granularity (i.e., only for non-locally updated objects) so does not suffer from wasted CPU resources on frequent snapshotting. Compared to the time-partitioning approach (TP-Grid), PGrid achieves up to 25% lower throughput, but does not suffer from stale and delayed query results and from difficult-to-set tuning parameters.

References

1. V. Akman, W. R. Franklin, M. Kankanhalli, and C. Narayanaswami. Geometric computing and the uniform grid data technique. *CAD*, 21(7):410–420, 1989.
2. E. Anderson and J. Tucek. Efficiency matters! *SIGOPS Oper. Syst. Rev.*, 44(1):40–45, 2010.
3. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10, 1995.
4. P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM TODS*, 8(4):465–483, 1983.
5. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
6. T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.
7. 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*, pages 181–190, 2001.
8. K. Chakrabarti and S. Mehrotra. Efficient concurrency control in multidimensional access methods. In *SIGMOD*, pages 25–36, 1999.
9. S. Chen, C. S. Jensen, and D. Lin. A benchmark for evaluating moving object indexes. *PVLDB*, 1(2):1574–1585, 2008.
10. 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.
11. J. Dittrich, L. Blunschi, and M. A. V. Salles. Indexing moving objects using short-lived throwaway indexes. In *SSTD*, pages 189–207, 2009.
12. J. Dittrich, L. Blunschi, and M. Vaz Salles. Movies: indexing moving objects by shooting index images. *GeoInformatica*, 15(4):727–767, 2011.
13. A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM TODS*, 30(2):492–528, 2005.
14. A. Fekete, E. O'Neil, and P. O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Record*, 33(3):12–14, 2004.
15. J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993.
16. J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks and degrees of consistency in a shared data base. In *VLDB*, pages 428–451, 1975.
17. S. Hwang, K. Kwon, S. Cha, and B. Lee. Performance evaluation of main-memory R-tree variants. In *SSTD*, pages 10–27, 2003.
18. Intel 64 and IA-32 Architectures Software Developers Manual. *Volume 3A: System Programming Guide, Part 1*. 2011.
19. C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B+-tree based indexing of moving objects. In *VLDB*, pages 768–779, 2004.
20. J. Jiang, H. Bao, E. Y. Chang, and Y. Li. MOIST: a scalable and parallel moving object indexer with school tracking. *PVLDB*, 5(12):1838–1849, 2012.
21. D. Kalashnikov, S. Prabhakar, and S. Hambrusch. Main memory evaluation of monitoring queries over moving objects. *Distributed and Parallel Databases*, 15(2):117–135, 2004.
22. K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *SIGMOD*, pages 139–150, 2001.
23. M. Kornacker and D. Banks. High-concurrency locking in R-trees. In *VLDB*, pages 134–145, 1995.
24. M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and recovery in generalized search trees. In *SIGMOD*, pages 62–72, 1997.
25. Kraftfahrt-Dundesamt (Federal Motor Transport Authority). *Anual Report*. Kraftfahrt-Bundesamt, Germany, 2009.
26. 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*, pages 608–619, 2003.
27. P. L. Lehman and s. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM TODS*, 6:650–670, 1981.
28. 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.
29. K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, pages 634–645, 2005.
30. V. Ng and T. Kameda. Concurrent access to R-trees. In *SSD*, pages 142–161, 1993.
31. L.-V. Nguyen-Dinh, W. G. Aref, and M. F. Mokbel. Spatio-temporal access methods: Part 2 (2003 - 2010). *IEEE Data Eng. Bull.*, 33(2):46–55, 2010.
32. S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In *MDM*, pages 7–16, 2011.
33. J. M. Patel, Y. Chen, and V. P. Chakka. Stripes: an efficient index for predicted trajectories. In *SIGMOD*, pages 635–646, 2004.
34. POSIX.1-2008. The open group base specifications, 2008.
35. R. Rastogi, S. Seshadri, P. Bohannon, D. W. Leinbaugh, A. Silberschatz, and S. Sudarshan. Logical and physical versioning in main memory databases. In *VLDB*, pages 86–95, 1997.
36. B. Salzberg. Grid file concurrency. *Information Systems*, 11(3):235–244, 1986.
37. 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.
38. 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*, pages 1150–1160, 2007.
39. Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: an optimized spatio-temporal access method for predictive queries. In *VLDB*, pages 790–801, 2003.
40. K. Tzoumas, M. L. Yiu, and C. S. Jensen. Workload-aware indexing of continuously moving objects. *PVLDB*, 1(2):1186–1197, 2009.
41. D. Šidlauskas, C. S. Jensen, and S. Šaltenis. A comparison of the use of virtual versus physical snapshots for supporting update-intensive workloads. In *DaMoN*, pages 1–8, 2012.
42. D. Šidlauskas, K. A. Ross, C. S. Jensen, and S. Šaltenis. Thread-level parallel indexing of update intensive moving-object workloads. In *SSTD*, pages 186–204, 2011.
43. D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen, and D. Šaulys. Trees or grids? Indexing moving objects in main memory. In *GIS*, pages 236–245, 2009.
44. D. Šidlauskas, S. Šaltenis, and C. S. Jensen. Parallel main-memory indexing for moving-object query and update workloads. In *SIGMOD*, pages 37–48, 2012.
45. O. Wolfson and H. Yin. Accuracy and resource consumption in tracking and location prediction. In *SSTD*, pages 325–343. 2003.
46. W. Wu and K.-L. Tan. iSEE: Efficient continuous k-nearest-neighbor monitoring over moving objects. In *SSDBM*, page 36, 2007.
47. X. Xiong, M. Mokbel, and W. Aref. SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654. IEEE, 2005.
48. M. Yiu, Y. Tao, and N. Mamoulis. The B^{dual}-tree: indexing moving objects by space filling curves in the dual space. *The VLDB Journal*, 17:379–400, 2008.

A The Cost of Frequent Snapshotting

By Definition 1, timeslice semantics guarantee zero staleness query results. The query staleness under stale-timeslice semantics depends on a tuning parameter that controls the snapshotting frequency, F_s. On average, one half of the updates that occur during $\frac{1}{F_s}$ time units are ignored when running under two-snapshot isolation. Denoting the total number of objects by N and the average time between any two consecutive updates of an object by T_o, we can compute the average staleness of a query as follows.

$$\frac{N/T_o \times \frac{1}{F_s} \times \frac{1}{2}}{N} = \frac{1}{2 \times F_s \times T_o} \quad (1)$$

Figure 18 illustrates the behavior of the above equation when varying the values of F_s (the x axis) and T_o (separate lines). Even with update rates as high as $T_o = 1$ s[7], query staleness below 1% remains feasible. However, this is at the cost of a very high snapshotting frequency. For example, to keep the query staleness below 1% for objects sending updates every 10 seconds, 5 snapshots per second are needed.

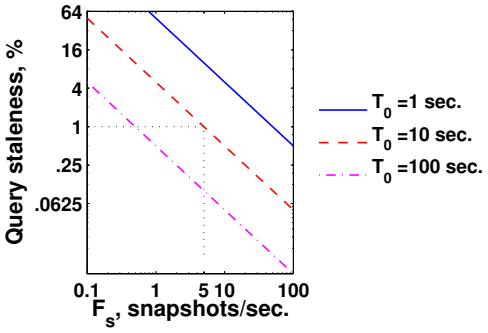


Fig. 18: Average query staleness based on Definition 1.

In main memory indexing, very frequent snapshotting, on the order of tens of snapshots per second, is feasible [42]. However, frequent snapshotting implies a substantial and unattractive waste of computing resources. For example, if one aims to maintain a maximum query staleness below 1%, snapshotting must be made whenever 1% of all objects are updated. The versions created for the remaining 99% of objects are superfluous. In practice, it is not trivial to set F_s optimally, as update rates can be highly skewed across time (peak hours versus night hours) [40].

B The Proof of Theorem 1

Theorem 1 *The* `rangeQuery` *algorithm (Algorithms 4–5), performed in the presence of concurrent* `update` *operations (Algorithms 1–3), returns results that satisfy freshness semantics.*

Proof We prove the theorem by examining all cases of Definition 2. Objects falling under cases 2c) and 2d) always satisfy the definition because the query algorithm is free to include or exclude such objects from the result set O. Objects falling under case 2b) are correctly excluded from O because the query algorithm considers adding an object to the result set only if that object has at least one of its index positions ($pos^\bullet$ or pos°) in the query region. This check is not shown explicitly in Algorithm 4, but it is assumed to be performed in Lines 2 and 10.

What remains is to prove that objects falling under cases 1) and 2a) of the definition are correctly processed by the algorithm. Case 1) requires that objects last updated before t_s should be included or excluded from the query based only on their current position pos°. Case 2a) requires that no object should be excluded from the query result set if the object was updated during the query processing time and both its previous as well as current positions are in the query region.

During the query processing time, a specific object o may be (i) not touched at all, (ii) locally updated with an atomic update, or (iii) physically moved in the index structure (more precisely, either $pos^\bullet$ or pos° is moved).

In the following, we cover all three scenarios for objects falling under cases 1) and 2a) of Definition 2. If o was not touched, $o.t_u < t_s$ and case 1) applies. Lines 6 and 7 of Algorithm 4 guarantee that only objects with $pos^\circ \in \Re$ are added to the result set. All such objects are added because all relevant cells are visited in Lines 2 and 10.

If o was updated locally then case 2a) applies. The object is included in the result set either in line 7 (pos°) or in line 9 ($pos^\bullet$).

The most challenging scenario occurs when an object is physically moved in the index structure during the time the query is processed. This can happen in two cases: when the object is updated non-locally or when it is the last object of the first bucket and it is moved to the place of a physically deleted object (Lines 6–7 of Algorithm 2). If the object is non-locally updated, case 2a) of the definition applies. The update leaves the object's previous position ($pos^\bullet$) in the index. So if the query misses the current position, it catches the previous one, as both of them are in the query region. Note that the query would miss the previous, logically deleted position if it was physically deleted during the query processing. This cannot happen as the physical deletion is carried out later during the next update of the object. According to the assumption that $t_e - t_s < T_o$ (see Section 3.4), the object can be updated only once during the query processing, so the next update will happen only after the query has finished.

Finally, if o is moved as the last object of the first bucket, both case 1) and case 2a) of the definition may apply. Observe that the query algorithm scans all buckets of a cell starting from the first bucket and that in each bucket, it scans entries backwards starting from the last entry (Algorithm 4). This guarantees that o cannot be moved from the unscanned part of the bucket list to the scanned part. So the query cannot miss o (but may see it twice). □

C The Proof of Theorem 2

Theorem 2 *The* `kNNQuery` *algorithm (Algorithms 5–6), performed in the presence of concurrent* `update` *operations (Algorithms 1–3), returns results that satisfy freshness semantics.*

Proof We examine all cases of Definition 3. While case 3) is trivial, we need to prove that objects falling under case 1) (case 2)) are always included in (excluded from) the result set O.

We start with an object o falling under case 1) and show that it is returned in *res* by Algorithm 6.

During the query processing, object o may be (i) not touched at all, (ii) locally updated with an atomic update, or (iii) physically moved in the index structure, which in turn can happen either due to a non-local update or because the object was moved as the last object of the first bucket. For our purposes, we divide these cases into two groups: a non-local update and all the remaining cases when the object stays in the same cell during the query processing.

We start by examining the latter cases first. Theorem 1 establishes that if the object stays in the same cell and the query scans that cell, it sees at least one of the positions of the object valid during the query processing interval (there can be one or two such positions).

[7] For instance, 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.

For brevity, we use the term closest (furthest) to mean closest to (furthest from) the query point q. Object o could be missed if the cell containing the object is not processed in the main loop of the algorithm. As the cells are processed in non-decreasing order of $dist(q,cell)$, the cell containing o will not be processed if $dist(q,cell) \geq cd$. Variable cd is updated only in line 12 of Algorithm 6. It starts from infinity and, due to the condition of line 8, can never increase. The lowest it can get is defined by the right side of the inequality in case 1) of Definition 3, which is the distance from q to the k-th closest position if, for *all* objects, the closest positions, among those valid during the query interval, are considered. Variable cd, on the other hand, is always assigned to the furthest position of the positions observed by the query of k unique objects, which is ensured by line 12. In addition, only object positions valid after ts are considered, as required by Definition 3 and ensured by line 7 of the algorithm.

Thus, cd will always be greater than the distance from q to o, which means that $dist(q,cell) < cd$ for a *cell* containing o. This in turn means that not only is the cell containing o scanned, but o passes the condition of line 8 and is added to *res*. Furthermore, once added, o will not be removed from *res* because case 1) of the Definition states that even if the furthest position of o is considered, its distance to q will be strictly smaller than the distance of the k-th closest object when all the closest positions of objects are considered. Thus, o cannot become the k-th object in *res* and so it cannot become a candidate for removal in line 11 of the algorithm.

Next, we examine the case where o is non-locally updated during the execution of the query. Case 1) of the definition guarantees that both $o.pos^{\bullet}$ and $o.pos^{\circ}$ are closer to q than the right side of the inequality. As detailed above, this means that both the cell containing $o.pos^{\bullet}$ and the cell containing $o.pos^{\circ}$ are scanned by the algorithm. As argued in the proof of Theorem 1, if both of these cells are scanned, $o.pos^{\bullet}$, $o.pos^{\circ}$, or both will be seen by the query. According to the argument above, one of them (the first seen by the query), will be included in *res* and will stay there until the end of the algorithm.

The second part of the proof concerns objects covered by case 2) of the definition. To prove by contradiction, we assume that an object falling under case 2) is returned in *res*. This means that according to lines 7 and 8 of the algorithm, there was o that had a position valid during $[t_s,t_e]$ such that $dist(q,o)$ was smaller than the value of cd at that time in the execution of the algorithm. That is, when o is added to *res*. Furthermore, because o is not removed from *res*, cd never gets lower than $dist(q,o)$, as cd is always equal to the k-th (furthest) object in *res* (see line 12 of the algorithm).

Case 2 of the definition says that even if the closest position of o valid during $[t_s,t_e]$ is considered, there are at least k objects such that all of their valid positions during $[t_s,t_e]$ are closer to q than $dist(q,o)$. Let us call the set of such objects K. We established that cd never gets lower than $dist(q,o)$ if o is returned in *res*. Thus, all the cells containing the positions of all objects from K valid during $[t_s,t_e]$ are scanned by the algorithm. Using the arguments from the proof of Theorem 1, we can state that the algorithm will see at least one of the valid positions for each of $x \in K$. Such positions will pass the test in line 8 of the algorithm, so all objects from K will be added to *res*. Furthermore, once added, such objects cannot be removed from *res* if o is not removed. This is because *res* is ordered on the distance and each $x \in K$ is always closer to q than o during $[t_s,t_e]$. Thus, at least k objects from K plus o will be in *res*, leading to a contradiction, as *res* never contains more than k objects (ensured by line 11 of the algorithm).

D Parallel Object Data Reads and Writes

To quantify the overheads of different ways to read/write a 128-bit value in an atomic manner, we conduct a micro-benchmark on the Intel Ivy Bridge processor (cf. Appendix E). We measure the CPU cycles needed per 128-bit read/write. Table 4 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.

Table 4: CPU cycles per 128-bit read/write.

	single-threaded		multi-threaded	
Method	read	write	read	write
multi-read/write	4	4	-	-
`pthread_mutex_t`	70	70	517	521
`pthread_rwlock_t`	111	108	1150	1158
`pthread_spinlock_t`	25	25	153	170
1-byte latching using CAS	24	24	134	155
OLFIT	5	69	262	260
SIMD	3	3	19	18

A simple multi-read/multi-write (combining two 64-bit reads and writes) consumes 4 CPU cycles and is correct only in the single-threaded case. For the multi-threaded case, we first profile the performance of the lock-based approaches using the three types of synchronization techniques defined in the Pthreads standard [34]. 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. 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 4). The multi-threaded case shows how performance degrades further due to cache coherency protocols [28]. Since the actual reads/writes are fast (4 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 implement 1-byte latches (or spin locks) using the atomic CAS instruction and use them instead of Pthread synchronization in our implementations (calls to lock/unlock objects and cells in Algorithms 1–5). As Table 4 shows, its performance is very similar to that of spin locks.

OLFIT reads show 5–20 times better performance than the lock-based reads in the single-threaded setting. However, OLFIT writes, due to extra operations (latch and unlatch using a relatively expensive atomic CAS instruction and version incrementing), are more expensive than writes using 1-byte latches or spin locks. 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.

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 (Table 1). Also, since single-object reads and writes are fast (4 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.

We found that SIMDized reads and writes are the most efficient, outperforming also the lockless multi-read/write in the single-threaded case. In the single-threaded case, it takes 3 cycles for both read and

Table 3: Experimental platforms.

	AMD Opteron 6176	Dual Intel X5650	Dual Intel E5-2670	Intel Core i7-3770
Architecture	Magny-Cours	Westmere	Sandy Bridge	Ivy Bridge
Linux Kernel	2.6.18	2.6.18	2.6.32	3.5.0
Compiler	GCC 4.1.2	GCC 4.1.2	GCC 4.4.6	GCC 4.7.2
Clock rate (GHz)	2.3	2.6	2.6	3.4
CPUs (sockets)	1	2	2	1
Cores (threads/core)	12 (1)	6 (1^9)	8 (2)	4 (2)
RAM (GB)	24	48	64	16
RAM type	DDR3-1333	DDR3-1333	DDR3-1600	DDR3-1600
L1 (data) cache (×cores)	64 KB	32 KB	32 KB	32 KB
L2 (unified) cache (×cores)	512 KB	256 KB	256 KB	256 KB
L3 (unified) cache	12 MB	12 MB	20 MB	8 MB
Cache line size	64 B	64 B	64 B	64 B

write, while in the multi-threaded case, due to the cache coherency protocol, the cost increases to 19/18 cycles per read/write.

E Experimental Setting

We study the performance on diverse multi-core platforms: a 12-core AMD Opteron 6176 (Magny-Cours), a dual 6-core Intel X5650 (Westmere) with 12 hardware threads[8], a dual 8-core Intel E5-2670 (Sandy Bridge) with 32 hardware threads, and a quad-core Intel Core i7-3770 (Ivy Bridge) with 8 hardware threads. 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. The characteristics of these machines are summarized in Table 3.

All indexing techniques were implemented in C++ and compiled with `g++` under the maximum optimization level. `Pthreads` and one-byte latches (see Appendix D) are used for parallelism.

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 [42], as all 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. This setting eliminates all other sources of contention and exposes the thread-level parallelism provided by the index structure and its operations. This makes the reported results independent of the chosen queue implementation and of how the load is distributed among the threads.

[8] Hyper-threading is disabled.