LSM-Based Storage and Indexing: An Old Idea with Timely Benefits

Sattam Alsubaiee
King Abdulaziz City for Science and Technology
Riyadh, Saudi Arabia
ssubaiee@kacst.edu.sa

Michael J. Carey, Chen Li
University of California, Irvine
CA 92697, USA
{mjcarey, chenli}@ics.uci.edu

ABSTRACT

With the social-media data explosion, near real-time queries, particularly those of a spatio-temporal nature, can be challenging. In this paper, we show how to efficiently answer queries that target recent data within very large data sets. We describe a solution that exploits a natural partitioning property that LSM-based indexes have for components, allowing us to filter out many components when answering queries. Our solution is generalizable to any LSM-based index structure, and can be applied not just on temporal fields (e.g., based on recency), but on any "time-correlated fields" such as Universally Unique Identifiers (UUIDs), user-provided integer ids, etc. We have implemented and experimentally evaluated the solution in the context of the AsterixDB system.

1. INTRODUCTION

Not long ago, traditional data warehouse systems were the norm for data analytics, and that technology was accessible only to large enterprises. The growth of social-networking data, however, combined with open source software platforms, has motivated smaller companies and organizations to collect and store huge amounts of data on a daily basis as well. Thanks to the recent advances in the Big Data space, those companies can now use Hadoop-based solutions to analyze the data and gain valuable insights that can help them sustain their businesses. With the evolving world of social media, however, it has quickly become evident that not all data-analytics problems can be solved efficiently with Hadoop, particularly due to its nature as a batch processing system. Location-based advertising, credit card fraud analytics, and recommendation systems are examples of applications that require real-time responses for which the speed of answering such queries in Hadoop is not sufficient. Therefore, new data platforms such as NoSQL systems and stream-processing systems have emerged to handle such use cases (e.g., MongoDB, HBase, Cassandra, BigTable, Spark, Storm, etc.). In many cases, companies have ended up using Hadoop-based solutions for long-running tasks, mostly for analyzing historical data, while using index-based (e.g., LSM-tree based NoSQL stores) and streaming-based solutions for short-running tasks that are more time-critical for their businesses.

GEORICH'15 , May 31 - June 04 2015, Melbourne, VIC, Australia
 ISBN 978-1-4503-3668-0/15/05...$15.00
DOI: http://dx.doi.org/10.1145/2786006.2786007.

One class of queries that require real-time answers and have a wide range of use cases are spatio-temporal query workloads favoring recent data. As an example, suppose a campaign manager for a US presidential candidate wants to know how potential voters are currently reacting to the candidates in a certain geographic area. A useful piece of information is the level of voters' interest in other rivals, as this is clearly valuable to the decision-making process of the campaign. We can formulate a spatial aggregation query to find tweets mentioning the names of other rivals that have been posted within the *last* day, group them on a spatial grid structure, and compute the number of such tweets for each cell in the grid. By doing this time-based spatial analysis, the campaign staff can gain an understanding of the current public opinion and make informed decisions such as broadcasting more political ads in certain areas.

In this paper, we study how to answer queries on recent data efficiently in the context of the AsterixDB Big Data Management System. AsterixDB has taken a different approach than other systems by combining a wide range of capabilities in a single unified system so as to provide better manageability, functionality, and performance, as opposed to gluing together multiple independent solutions. One novel feature of AsterixDB is wholly adopting LSM-trees as the underlying technology for all of its internal data storage and indexing. We show how to utilize a natural partitioning property that LSM-based indexes have for components to provide near real-time responses to query workloads that favor recent data.

The rest of this paper is organized as follows. First, we provide a brief introduction to AsterixDB. Next, we explain our solution, which exploits the multi-component nature of LSM-based indexes to provide near real-time performance for queries on recent data. Finally, we briefly present results from an experimental study that we conducted to evaluate the solution.

2. THE ASTERIXDB SYSTEM

In this section, we briefly introduce AsterixDB [2], including its data model, query language, and storage engine.

2.1 Data Model and Query Language

Consider the US election example mentioned in Section 1. We will use this example to briefly show the capabilities of AsterixDB, including its spatial support. The Asterix Data Model (ADM) is based on ideas from JSON extended with additional primitive types as well as type constructors borrowed from object databases. Figure 1 shows the use of ADM to model a Twitter messages dataset.

Data types in ADM are *open* by default; instances will conform to the type specification but may have extra fields with names and types that vary from one instance to the next. The "?" in the `sender-location` field means that the presence of this geospatial field is anticipated but optional. The `hashtags` field is a col-

```
create type TwitterUserType {
    screen-name: string,
    lang: string,
    friends_count: int32,
    statuses_count: int32,
    name: string,
    followers_count: int32
};
create type TweetType {
    tweetid: int64,
    user: TwitterUserType,
    sender-location: point?,
    send-time: datetime,
    hashtags: {{ string }},
    message-text: string,
    userid: int32
};
create dataset Tweets(TweetType) primary key tweetid ;
```

Figure 1: Metadata definition for the running example.

lection (multiset or unordered list) of primitive string values, and user information is modeled as a nested record of another type. As shown in Figure 1, a dataset called `Tweets` is created to store data instances conforming to the `TweetType` data type, where the primary key is the `tweetid` field. (An AsterixDB dataset is loosely analogous to a table in the relational world.)

The Asterix Query Language (AQL) is a declarative query language that draws on ideas from XQuery. Suppose we are just a few days away from the US presidential election that is going to be held on November 8th, 2016. A campaign manager could use a GUI to submit a spatial aggregation query to AsterixDB, such as the query shown in Figure 2, to see how potential voters are reacting to the candidate "Hillary Clinton", *recently*, in the swing state of Ohio. This query spatially aggregates election-related tweets. (Its variables are denoted by "$" identifiers.) It starts by constraining tweets to being within a bounding rectangle inside Ohio, a datetime window of the last day, and containing the hashtag "Hillary Clinton". The `spatial-cell` function determines which grid cell each tweet belongs to. This function receives the location of the tweet, the origin of a bounding rectangle for the grid, and the latitude and longitude increments that specify the resolution of the grid. It returns the cell (represented by a rectangle) that the tweet belongs to. Those tweets are then grouped according to their containing grid cells. Finally the `count` function is applied to each group of tweets to return the final answer in the form of pairs of cells and the number of tweets (that satisfy the predicates) in the corresponding cell. Figure 3 shows a color-coded density grid on the map that visualizes the results of a typical spatial aggregation query using the Google Maps API.

```
for $tweet in dataset Tweets
let $searchHashTag := "Hillary Clinton"
let $leftBottom := create-point(38.52,-84.78)
let $rightTop := create-point(41.94,-80.48)
let $latResolution := 3.0
let $longResolution := 3.0
let $region := create-rectangle($leftBottom,$rightTop)
where spatial-intersect($tweet.sender-location, $region)
and $tweet.send-time > datetime("2016-11-05T00:00:00Z")
and (some $hashTag in $tweet.hashtags
  satisfies ($hashTag = $searchHashTag))
group by $cell := spatial-cell($tweet.sender-location,
  $leftBottom, $latResolution, $longResolution)
  with $tweet
return { "Cell": $cell, "NumTweets": count($tweet) }
```

Figure 2: A spatial aggregation query over tweets that were generated by Ohio state users, close to the US presidential election in 2016, containing the hashtag "Hillary Clinton".

2.2 Storage Management

Datasets are managed by AsterixDB as partitioned LSM-based

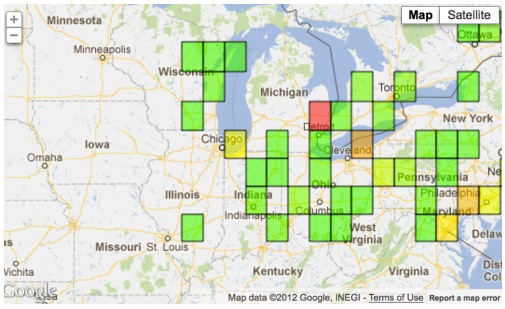

Figure 3: A visualization of the results of a spatial aggregation query. The color of each cell indicates the tweet count.

B$^+$-trees with optional LSM-based secondary indexes [3]. LSM-trees [7] are well-known for providing superior performance for insert-intensive workloads by batching updates into a component of the index that resides in main memory – an *in-memory component*. When the space occupancy of the in-memory component exceeds a specified threshold, its entries are *flushed* to disk forming a new component – a *disk component*. As disk components accumulate on disk, they are periodically merged together subject to a *merge policy* that decides when and what to merge. The benefit of LSM-trees comes at the cost of possibly sacrificing read efficiency, but, as shown in [8], these inefficiencies can be mostly mitigated.

AsterixDB has adopted a framework for converting a class of indexes (including conventional B+ trees, R trees, and inverted indexes) into LSM-based secondary indexes, allowing higher data ingestion rates. In fact, for certain index structures such as the LSM R-tree, our results [3] have shown that an LSM-based version of an index can be made to significantly outperform its conventional counterpart for *both* data ingestion and query speed.

3. LSM-BASED FILTERS FOR ACCELERATING QUERIES

In this section, we present a technique that leverages the structure of an LSM-based index, including the LSM R-tree, to accelerate queries. Our technique works best with a monotonically increasing sequence of values, e.g., time-correlated fields such as UUID and datetime fields, making it a perfect fit for providing near real-time performance for recency queries such as the time-based spatial aggregation query of Figure 2.

3.1 Basic Idea

Since an LSM-based index naturally partitions data into multiple disk components, it is possible, when answering certain queries, to exploit partitioning to only access some components and safely filter out the remaining components, thus reducing query times. This can be achieved by augmenting each disk component with extra information (called a "filter" hereafter) about one or more other fields of the record (called the filter's key hereafter). Hence, the index can filter out entries based on two dimensions, one based on the original index key(s), and the second based on the filter's key.[1]

To support filtering, each LSM disk component has an associated filter record that maintains the minimum and maximum filter key values for the records in the component. Then, when accessing the index to answer a query, the index lookup operation can first leverage the associated filter records to prune components that do

[1]The current release of AsterixDB (Release 0.8.6) supports the creation of single-field filters. Allowing multi-field filters is a straightforward extension and it is planned for a future release.

not match the filter's predicate. The normal search on the index need only be performed for those components that survive the filtering process. Notice that filters can be used with both primary *and* secondary indexes for additional pruning power.

One of the main use cases for LSM-based filters is to use them to index time-correlated fields or monotonically increasing sequences, such as datetime fields. In this case, the filters on the disk components are likely to have disjoint minimum and maximum values, making them very effective for pruning. Next, we provide two examples that show the benefits of using filters with LSM indexes.

Example 1: Suppose that users are interested in retrieving recent tweets from the `Tweets` dataset posted by specific users based on their sending time (e.g., tweets posted by "John Smith" in the last day). Figure 4 shows the AQL query for this example. We can create a filter on the `send-time` field of the primary index and then AsterixDB can utilize the components' filter records to quickly prune components that do not match the temporal predicate.

```
for $tweet in dataset Tweets
where $tweet.send-time > datetime("2015-05-14T00:00:00Z")
and $tweet.user.name = 'John Smith'
return $tweet
```

Figure 4: A query that returns all the tweets posted by "John Smith" in the last 24 hours (assuming the current date is May 5th, 2015).

Example 2: Consider the spatial aggregation query, shown in Figure 2. We can maintain a secondary LSM R-tree index on the `sender-location` field and create a filter on the `send-time` field of the primary and secondary indexes. When answering the spatial aggregation query, AsterixDB can first access the LSM R-tree, using the index components' filter records to quickly filter out those components that do not match the temporal predicate of the query, and then search the remaining R-tree components that survive the filtering process. Similarly, when using the resulting primary keys to probe the primary index, the lookup operation can filter out all older disk components of the index and probe only those components that satisfy the temporal predicate.

Clearly, LSM-based filters can help improve the performance of queries by pruning as many records as possible in an early stage. They also can improve data ingestion performance by not requiring a separate secondary index in some scenarios (as in the first example). In addition, they can potentially improve data ingestion performance under concurrent queries by reducing contention on system resources (CPU and I/O) due to component filtering.

3.2 LSM-based Filters in AsterixDB

In this section, we discuss the implementation of LSM-based filters in AsterixDB. We first describe how to declare dataset filters at the logical level. We then explain the changes that we made to the system's internals to incorporate filters in AsterixDB's indexes.

3.2.1 User Model

We have added support for LSM-based filters to all of AsterixDB's index types (LSM B$^+$-trees, LSM R-trees, and LSM inverted indexes). To enable the use of filters, the user must specify the filter's key when creating a dataset, as shown in Figure 5. Filters can be based on any totally ordered datatype (i.e., any field that can be indexed using a B$^+$-tree), such as integers, doubles, floats, UUIDs, datetimes, etc.

The name of the filter's key field is persisted in the "dataset" dataset (which is the metadata dataset that stores the details of each dataset in an AsterixDB instance) so that DML operations against

```
create dataset Tweets(TweetType) primary key tweetid
with filter on send-time;
```

Figure 5: Creating a dataset of tweets with a filter on the `send-time` field.

the dataset can recognize the existence of filters and can update them or utilize them accordingly. Creating a dataset with a filter in AsterixDB implies that the primary and all secondary indexes of that dataset will maintain filters on their disk components. Once a filtered dataset is created, the user can use the dataset normally (just like any other dataset). AsterixDB will automatically maintain the filters and leverage them to efficiently answer queries whenever possible (i.e., when a query has predicates on the filter's key).

3.2.2 Maintaining the Filters

In an LSM index, there are three possible methods where a new disk component is created: through a flush, merge, or fresh bulk-load operation. To guarantee the correctness of a dataset's filters, each of these operations must ensure that the filter associated with a new disk component has the minimum and maximum filter key values of the records contained therein. In the following, we show how each of those operations maintains these values efficiently:

1. Flush: Each index incrementally maintains the minimum and maximum filter key values for the records contained in its in-memory component. Those values must be updated, as needed, with each insert and delete operation against the index. When the in-memory component is flushed to disk, its filter record is also flushed with it to disk.

2. Merge: The minimum and maximum values for the resulting component's filter record are taken from the smallest and largest values of all components participating in the merge.

3. Bulk-load: Each added entry simply updates the filter record's minimum and maximum values as needed.

The leaf node entries in secondary indexes in AsterixDB are of the form $e = \langle SK, PK \rangle$, where SK is the secondary key and PK is the associated primary key. Although this form is still maintained in the presence of filters, we had to change the insert, delete, and load plans for indexes with filters so that the filter key values are also passed to the index with each record for the purpose of updating its components' filter records.

3.2.3 Query Processing

We have added a new rewrite rule to the AsterixDB rule-based optimizer that checks whether or not a query can be optimized to use filters. The rule first checks to see if the queried dataset has a filter. If so, then it analyzes the query in the hope of finding an applicable query predicate. In particular, the rule searches for predicates on the filter's key that use one of the following operators: $>$, $>=$, $<$, $<=$, and $=$. If it finds such predicates, it modifies the plan so that the filter's predicates are passed to all the indexes of the dataset that appear in the query plan (in conjunction with other predicates that are normally used to search the indexes).

At runtime, the filter's predicates will be used to prune components that do not match. Note that the implementation of AsterixDB's LSM indexes was designed from the start to allow each index operation (e.g., search) to choose the components that it needs to perform its logic against, and this is where component filtering takes place. In other words, the search cursors for LSM indexes did not have to change to support filters. Once the components

not matching the filter's predicates have been filtered out, a normal search on the index is performed for the remaining components.

3.2.4 *Effect of Merge Policies*

Merge policies play a key role with respect to the data ingestion and query performance of LSM indexes. This role is even more important for LSM-based filters to be effective. Here we discuss the tradeoffs of the various AsterixDB merge policies (constant, prefix, and no-merge policies [3]) and introduce a new merge policy that further improves query performance when using filters.

To understand the effect of merge policies on filters, let us revisit a similar concept that has been heavily used in many database systems, which is table partitioning. In these systems, partitioning is mainly used to mitigate the impact of table scans. A table can be divided into smaller partitions based on a partitioning key. Queries with predicates on the partitioning key can then prune those partitions that do not satisfy the predicates, thus reducing query times. Similarly, when using filters, each LSM disk component is essentially treated as a partition. Thus, the number and size of disk components can greatly impact query performance.

Generally speaking, having very few disk components will reduce a filter's pruning power for highly selective queries on the filter's key. On the other hand, less selective queries on the filter's key will perform best when the number of disk components is minimal. The *constant* merge policy tries to keep as few disk components as possible by constantly merging all the disk components into a single disk component. This has been shown in [3] to have a negative impact on ingestion performance since merges are CPU and I/O intensive operations. In addition, the constant merge policy will also have a negative impact on the performance of queries when using filters since the chances of accessing all the disk components becomes very high, eliminating the benefits of filters.

In contrast, when used in conjunction with filters, AsterixDB's *no-merge* policy can provide excellent performance for selective queries on the filter's key; this is due to the high chance of pruning many of the components. However, similar to the constant merge policy, the no-merge policy has proven to provide bad ingestion performance due to the time spent searching the many disk components before each primary key insert to enforce key uniqueness, making the no-merge a bad choice for write-intensive workloads.

The AsterixDB *prefix merge* policy, which relies on component sizes and the number of components to decide which components to merge, has proven to provide excellent performance for both ingestion and queries. However, when evaluating our filtering solution with the prefix policy, we observed a behavior that can reduce filter effectiveness. In particular, we noticed that under the prefix merge policy, the disk components of a secondary index tend to be constantly merged into a single component. This is because the prefix policy relies on a (single) size parameter for all of the indexes of a dataset. This parameter is typically chosen based on the sizes of the disk components of the primary index (e.g., 1GB in our experiments), which tend to be much larger than the sizes of the secondary indexes' disk components. This difference caused the prefix merge policy to behave similarly to the constant merge policy (i.e., relatively poorly) when applied to secondary indexes. Consequently, the effectiveness of filters on secondary indexes was greatly reduced under the prefix-merge policy, but they were still effective when probing the primary index.

Based on this behavior, we developed a new merge policy, an improved version of the prefix policy, called the *correlated-prefix* policy. The basic idea of this policy is that it delegates the decision of merging the disk components of *all* the indexes in a dataset to the primary index. When the policy decides that the primary index needs to be merged (using the same decision criteria as for the prefix policy), then it will issue successive merge requests to the I/O scheduler on behalf of all other indexes associated with the same dataset. The end result is that secondary indexes will always have the same number of disk components as their primary index under the correlated-prefix merge policy. This has improved query performance, as we will see in Section 4.5, as disk components of secondary indexes now have a much better chance of being pruned.

4. EXPERIMENTS

This section briefly presents the results of an experimental evaluation, including queries with temporal and spatial predicates, of the filtering solution designed and implemented in AsterixDB. We refer interested readers to [1] for more detailed experimental results.

4.1 Data Generation

We used AsterixDB's data feeds [4] to populate a dataset using streams of synthetic tweets, with an average tweet size of 1KB. The generated tweets conform to the type definition shown in Figure 1. We generated tweets with monotonically increasing ids, and chose user ids randomly with replacement from the range [0-4999]. All the generated tweets had geographic locations that were generated randomly from a distribution similar to the one described in [3]. The `send-time` value of each tweet was populated with the exact time of actual tweet generation.

4.2 Machine and Parameter Configurations

We used two machines[2] to conduct our experiments. The first one generated synthetic tweets to send over the network to a second machine hosting a single-machine AsterixDB instance for ingestion. The latter was an IBM server with a 4-core Xeon 2.27 GHz CPU, 12GB of main memory, and four 10,000 rpm SATA drives. We dedicated one disk to the transaction log manager, using the other three disks as data storage disks for separate partitions of the `Tweets` dataset and their associated secondary index partitions. Each dataset was given 1GB of memory for its in-memory component(s). Unless otherwise specified, we used AsterixDB's prefix merge policy to manage the disk components of each datset's indexes.

4.3 Query Generation

For our experiments, we ingested a total of 70GB of synthetic tweets into the targeted dataset. We then stopped data ingestion and issued queries with multiple predicates on different fields, with one predicate being based on time-recency to mimic a query workload that is targeting recent data. This predicate was based on the largest ingested `send-time` value. For example, if its largest ingested value was `datetime("2015-04-26T10:10:00Z")`, and the goal was to retrieve all tweets ingested in the last 1 minute, the predicate would be "retrieve all tweets with a `send-time` larger than `datetime("2015-04-26T10:09:00Z")`". We used the approach described in [3] to generate random spatial predicates that provide result sets with similar cardinalities.

We experimented with recency predicates with varying selectivities ranging from a predicate that only matches the most recent data (e.g., the last 8 seconds) all the way to a predicate that matches all of the ingested data. To do that, we chose an initial time unit (e.g., 8 seconds) that served as the smallest recency predicate width (e.g., all tweets ingested in the last 8 seconds). We then multiplied this time unit by different factors to increase the window size of the

[2] Since filters are purely a node-level storage optimization, scale-out testing on a multi-machine cluster was not required.

recency predicate. For those experiments that used an initial time unit of 8 seconds, we used the following time-unit multipliers: 1, 6, 42, 180, 540, and 2160. Thus, the corresponding recency predicates were: last 8 seconds, last 48 seconds, last 5.6 minutes, last 24 minutes, last 72 minutes, and last 288 minutes. Unless otherwise specified, we used an initial time unit of 8 seconds for all the experiments. The query times shown are averaged across 200 random queries using different time unit factors.

We plotted the results using histogram charts in log scale, where the X axis represents the different time-unit factors and the Y axis represents the average query time.

4.4 Query Performance

Figure 6 shows the average query time (log scale) for spatial aggregation queries, similar to the one shown in Figure 2 (excluding the textual predicate), against two datasets that are both using a secondary LSM R-tree on the field `sender-location`; one of them was also utilizing a filter on the field `send-time`.

The results show that the dataset that had a filter on `send-time` benefited greatly when the recency predicate was selective due to the pruning power of filters. For instance, for the time-unit factor one, all disk components of the corresponding indexes were pruned, resulting in searching only the in-memory component of the index. However, as we decreased the selectivity of the recency predicate by using larger time-unit factors, the effectiveness of filters decreased. This is expected since the less selective a recency predicate is, the more disk components that satisfy the predicate and thus must be searched. That being said, even for the extreme case when the recency predicate is unselective, filters do not hurt query performance and perform as if the filters did not exist. This can be seen when using the time-unit factor 2160, where all the records satisfy the recency predicate.

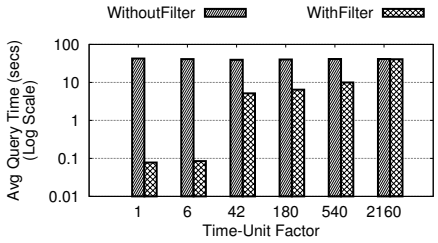

Figure 6: Average time to answer spatial aggregation queries when using a dataset with a secondary LSM R-tree on `sender-location` and a dataset with both a secondary LSM R-tree on `sender-location` and a filter on `send-time`.

4.5 Impact of Merge Policies

Next, we explored the impact of using different merge policies with filters. Each dataset had a secondary LSM B$^+$-tree on `userid`. Since the handled TPS when using a different merge policy varies widely (the total time for ingesting 70GB of tweets was 110 minutes versus 26 hours for the prefix versus no-merge policies, respectively), the selectivity of recency queries using the same time-unit factor would also vary widely. Thus, here we added a new field called `integer-send-time` to the `TweetType` definition and used it as the filter's key (it was populated with monotonically increasing integer values). This way, we were able to isolate the impact of TPS variance on the selectivity of queries, allowing for an accurate comparison between the different merge policies. We also used an initial time unit of 32,000 for all of the experiments in this section. (As an example, for the time-unit factor 540, the corresponding recency predicate would ask for all tweets with `integer-send-time` values that are 540 x 32000 = 17280000 smaller than the largest ingested `integer-send-time` value.)

Figure 7 shows the average query time (log scale) for answering recency queries, similarly to the query shown in Figure 8, for datasets using different merge polices. All the datasets provided identical average query time for the smaller time-unit factors, 1 and 6, because all disk components were pruned by the filters since they did not match the recency predicates.

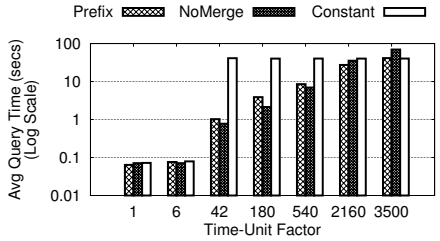

Figure 7: Average query time when using constant, no-merge, and prefix merge policies on a dataset with both a secondary LSM B$^+$-tree on `userid` and a filter on `integer-send-time`.

```
let $count := count (
    for $tweet in dataset Tweets
    where $tweet.integer-send-time > 105477969
    and $tweet.userid = 3319
    return $tweet
)
return {"NumTweets": $count}
```

Figure 8: One example of an aggregation query that we used in our experiments to compare the impact of different merge polices on filters.

Clearly, the performance differences are huge (log scale) when the recency predicates are less selective and, as a result, disk components must be accessed. The constant merge policy always provided the same high average query time for the larger factors. This is because it had a single disk component for both the primary and secondary indexes at the end of the ingestion period, and thus, the pruning power of the filters was lost.

Both the no-merge and prefix policy provided query times that are relative to the number of accessed disk components. For the middle time-unit factors (i.e., 42, 180, and 540), the prefix policy slightly suffered from having only three disk components for the secondary index, reducing the effectiveness of its filters. The index had only three disk components since, as described in Section 3.2.4, this policy tends to behave similarly to the constant policy when using large values for its size parameter. On the other hand, in the case of the no-merge policy, the queries had to touch fewer secondary index entries due to the many disk components that were pruned, and as a result there were much fewer primary index probes. However, as the selectivity of the recency predicate decreased, the no-merge policy had to pay for accessing many smaller disk components. This can be seen in the case of the largest two time-unit factors, where 158 and 242 out of 242 disk components (for both the secondary and primary indexes) had to be accessed for the 2160 and 3150 time-unit factors, respectively.

Figure 9 (log scale) shows how the new correlated-prefix merge policy combines the benefits of both the prefix and no-merge polices. For the middle time-unit factors (i.e., 42, 180, and 540), the

correlated-prefix policy behaved similar to the no-merge policy in the sense that it benefited from having multiple disk components for the secondary index. For the larger time-unit factors, 2160 and 3500, it behaved in a manner similar (with minor overhead) to the prefix policy in the sense that it does not have too many smaller disk components (there were 22 secondary index disk components).

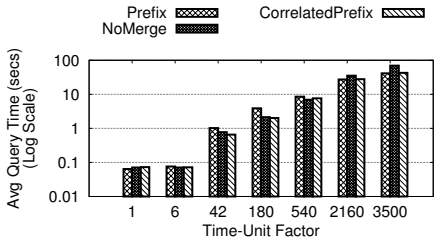

Figure 9: Average query time when using the correlated-prefix compared to the no-merge and prefix policies on a dataset with both a secondary LSM B$^+$-tree on `userid` and a filter on `integer-send-time`.

Figure 10 (again log scale) shows how the LSM R-tree behaved when using the correlated-prefix policy to answer queries similar to the one shown in Figure 2 (excluding the textual predicate). We see that under the correlated-prefix policy, the selective recency predicates benefited from having multiple smaller R-tree components. Clearly, the performance differences here are larger compared to those for secondary LSM B$^+$-tree. This is due to the fact that searching an R-tree is more expensive than searching a B$^+$-tree. For very large time-unit factors, however, the queries did not benefit from the filters. In this case, having very few disk components can help query performance, as can be seen in the case of the prefix policy for the time-unit factor 3500.

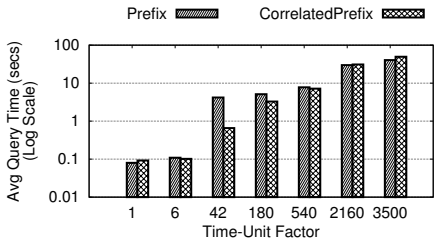

Figure 10: Average query time for answering spatial aggregation queries when using the prefix and correlated-prefix merge policies on a dataset with both a secondary LSM R-tree on `sender-location` and a filter on `integer-send-time`.

In addition to query performance, it is worth mentioning that data ingestion performance was not hampered by the filters since they are very cheap to maintain. In fact, when using the correlated-prefix policy, the data ingestion time was improved by around 10% due to performing fewer secondary index merge operations, and, as a result, reducing the system's CPU and I/O contention.

5. RELATED WORK

Using LSM-trees for indexing fast data ingestion was intrduced in [7]. Different variations of the LSM-tree were then suggested, e.g., [5, 8]. Muth et al. [6] have introduced an index structure called LHAM for transaction-time temporal data. They used an LSM-like structure to index record versions in two dimensions: one dimension is the conventional record key and the other is the timestamp of the record version. The record version is obtained at the time of insertion, representing the transaction timestamp. The outcome is that data is partitioned into successive components based on the timestamps of the record versions. Queries with temporal predicates only need to access components that satisfy the predicates; the remaining components can be skipped. HBase uses a similar idea by tagging its LSM disk components with the minimum and maximum timestamps of the records contained in the component, which can be used during query time for effective pruning.

Our work differs from LHAM and HBase in the following ways:

1) We do not limit the use of filters to transaction timestamps. Instead, when creating a dataset, a user may create a filter on any totally-ordered datatype field of the record (e.g., integer, double, datetime, UUID, etc.).

2) We apply filters not only on disk components of the primary index, but also on the disk components of secondary indexes, leading to significant additional pruning power.

3) We have studied the effect of different merge policies on query performance when using our generalized filters and experimentally demonstrated their tradeoffs and benefits.

6. CONCLUSIONS

In this paper, we have presented an efficient solution for handling spatio-temporal query workloads that favor recent data, allowing the AsterixDB system to provide near real-time performance for recency queries. We have designed and implemented filters for all of AsterixDB's index types, including LSM B$^+$-tree, LSM R-tree, and LSM inverted indexes for textual data. Our solution exploits the fact that LSM indexes partition data into successive disk components based on freshness. By maintaining filters on disk components of LSM indexes, we were able to reduce recency query response times by up to 99% (for very selective queries that only access the in-memory component of the index). Our experimental evaluation explored the impact of LSM merge policies and showed the benefits of our new correlated-prefix policy.

Acknowledgements This work was supported by a graduate fellowship from KACST. It was also supported by NSF projects IIS-0910989, CNS-1305430, and IIS-1447720. Industrial support came from Facebook, Google, HTC, and Oracle Labs.

7. REFERENCES

[1] S. Alsubaiee. *Spatial Indexing in the Era of Social Media*. Ph.D. thesis, UC Irvine, 2014.

[2] S. Alsubaiee et al. AsterixDB: A scalable, open source BDMS. *VLDB*, 2014.

[3] S. Alsubaiee et al. Storage management in AsterixDB. *VLDB*, 2014.

[4] R. Grover and M. J. Carey. Data ingestion in AsterixDB. *EDBT*, 2015.

[5] C. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. *The VLDB Journal.*, 16(4), 2007.

[6] P. Muth et al. Design, implementation, and performance of the LHAM log-structured history data access method. *VLDB*, 1998.

[7] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4), 1996.

[8] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. *SIGMOD*, 2012.