Noname manuscript No.
(will be inserted by the editor)

High-throughput Publish/Subscribe on top of LSM-based Storage

Mohiuddin Abdul Qader · **Vagelis Hristidis**

the date of receipt and acceptance should be inserted later

Abstract State-of-the-art publish/subscribe systems are efficient when the subscriptions are relatively static – for instance, the set of followers in Twitter – or can fit in memory. However, now-a-days, many Big Data and IoT based applications follow a highly dynamic query paradigm, where both continuous queries and data entries are in the millions and can arrive and expire rapidly. In this paper we propose and compare several publish/subscribe storage architectures, based on the popular NoSQL Log-Structured Merge Tree (LSM) storage paradigm, to support high-throughput and highly dynamic publish/subscribe systems. Our framework naturally supports subscriptions on both historic and future streaming data, and generates instant notifications. We also extend our framework to efficiently support self-joining subscriptions, where streaming pub/sub records join with past pub/sub entries. Further, we show how hierarchical attributes, such as concept ontologies, can be efficiently supported; for example, a publication's topic is "politics" whereas a subscription's topic is "US politics." We implemented and experimentally evaluated our methods on the popular LSM-based LevelDB system, using real datasets, for simple match and self-joining subscriptions on both flat and hierarchical attributes. Our results show that our approaches achieve significantly higher throughput compared to state-of-the-art baselines.

Keywords Log-Structured Merge Tree · LevelDB · Publish/Subscribe · Self-join subscription · Dewey · Big Data · Internet of Things · Continuous Lookup Queries

1 Introduction

In this age of big data and Internet of Things (IoT), large amounts of data are generated, stored, and used by a diverse set of entities – devices, vehicles, buildings,

Mohiuddin Abdul Qader · Vagelis Hristidis
Department of Computer Science & Engineering
University of California Riverside, California, USA
E-mail: {mabdu002,vagelis}@cs.ucr.edu

software, and sensors. It is challenging to efficiently ingest, manage, read and deliver the generated data to millions of users or entities in real time. Publish/Subscribe systems are used in many applications, such as social networks, messaging systems, and traffic alerting systems.

As a running example application, consider users who are driving and subscribe to nearby traffic or other incidents (accident, crime, roadwork, fire, natural disaster, protest etc). Every time a user moves to a new location (i.e., a geospatial cell) they need to subscribe to events/publications in the new location for a time duration starting from the near past to the near future – e.g., to know what happened in the last one minute and what will happen in the next one minute until the user moves to another cell. These subscriptions are *highly dynamic* as they come and go every few seconds. At the same time, users are publishing incidents. As millions of users are moving and subscribing to events, these large streams of subscriptions and publications must be stored and managed efficiently. Google's spatial notifications closely follow this moving subscriber example. Here, when subscribers go to a store, Google pushes to them, as a mobile notification, the map of the store, specials deals, and so on. As another application, an airplane continuously queries for data in its path such as turbulence, wind, air pressure, etc.

Challenges and requirements Figure 1 shows different layers in a typical pub/sub

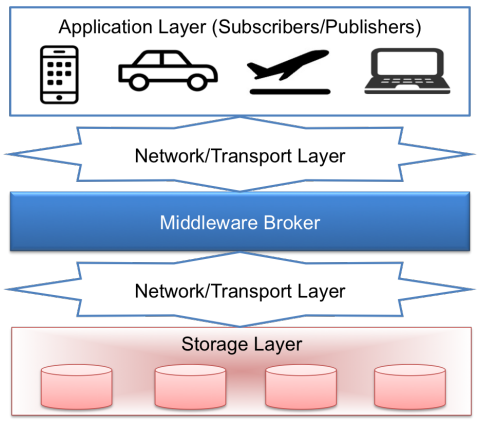


Fig. 1: Different layers in a typical Pub/Sub system. This paper focuses on the red-highlighted storage layer

system. Here we can see that subscriptions and publications can be generated from a variety of devices in application layer. The application communicates through network/transport layer with a middleware broker which notifies subscribers for a matching publication. A key component of a Publish/Subscribe system is its storage layer, which stores both the subscriptions and the publications. The broker talks with the storage layer to store/retrieve these publications/subscriptions. Our paper focuses on the storage layer of a pub/sub system.

As described in detail in Section 2, the storage modules of existing Publish/Subscribe systems have several *limitations*, which make them inadequate for modern IoT applications. First, the number of subscriptions (continuous queries) is assumed to be relatively small to fit in main memory, which may not always be true. Second, the subscriptions are viewed as relatively static, for example, a user infrequently modifies her following list in Twitter. This is not the case in applications such as traffic alerting or aviation where vehicles or airplanes subscribe to different cells of interest every few seconds. Third, most of the systems support only queries on future data. In contrast, an airplane may need to know the conditions in an area in the last few minutes and the next few seconds, so a combination of past and future data may be desired. The publications may also have an expiration time; for example, a snow storm warning might have a certain time limit.

A *baseline* solution to realize a Publish/Subscribe system, which we experimentally evaluate, is to maintain a list of subscriptions (queries) and periodically submit them as *repetitive queries* on the publications database. A similar approach is currently used in the AsterixDB BAD system [9], where such subscriptions are referred as repetitive channels. A drawback with this approach is that one cannot get notification at the exact time of an incident. Also, this solution wastes resources as the same query will keep getting submitted even if no new matching publications exist. Finally, some publications with short expiration time may be completely missed.

To summarize, our goal is to design and build a Publish/Subscribe storage system with the following properties:

1. Scale to millions of dynamically changing subscriptions and publications per minute, that is, both subscriptions and publications arrive and expire at a rapid pace.
2. After a new publication, immediately identify and notify matching subscribers (as in traditional database triggers, discussed in Section 2), that is, not follow a periodic check paradigm.
3. Subscriptions or publications may have validity time periods. Subscriptions may request past data in addition to future data.
4. The subscriber should assume that all the matching publications should reach her, that is, there is no data loss, which may occur with periodic check systems.

Contribution To support these properties we present an efficient storage framework on top of LSM-based NoSQL databases. We argue that NoSQL databases – such as Cassandra [26], Voldemort [15], AsterixDB [6], MongoDB [3] and LevelDB [2] – provide the right primitives –fast write throughput and fast lookups on the primary key – on top of which we can design an efficient Publish/Subscribe storage system that satisfies the above properties. In our paper, a database refers to a single LSM-based key-value store, which can be viewed as the equivalent to a table in a relational database. We built our prototypes on top of LevelDB, which is a popular and open-source LSM-based key-value store from Google. These LSM-based key-value stores as well as our databases contain both memory components(i.e. memtable) and disk components (i.e. SSTable).

We first present an approach based on two separate databases (*TwoDB*), one for subscriptions (queries) and one for publications (data records). The advantage of this

approach is that it is simple to implement, as it only requires little or no change to current LSM storage modules. We show that despite its simplicity, it is efficient compared to a baseline based on repetitive queries (*RepQueries*) mentioned above. However, this two-database approach suffers from a high rate of random disk accesses when the subscriptions not only match publications but also past subscriptions – e.g., return publications that have not been already returned by a past subscription. To efficiently answer such complex subscriptions, which we refer as *self-joining subscriptions*, we propose a novel *DualDB* approach where both queries and data entries are stored in the same keyspace of the same database, which leads to much fewer random accesses.

We also propose techniques to extend these storage architectures to handle hierarchical attributes. For example, a publication may specify a large affected area (higher level bounding rectangle in a spatial hierarchy), whereas a subscription may only be interested in a smaller spatial cell. Similarly, a publication may specify a specific topic like "soccer", whereas a subscription may specify a more general topic like "sports."

In addition to the repetitive queries baseline, we compare our approaches to a state-of-the-art pub/sub system, Padres [16]. We chose Padres because it is open source and is easy to modify to fit our use-case. Padres supports querying on historic/past data, and uses the PostgreSQL database system. We show that Padres does not scale well with the number of subscriptions. Specifically, the system runs out of memory and crashes very quickly, because it uses an in-memory module to manage the subscriptions.

To summarize, we make the following contributions in this paper:

- We propose and implement two novel approaches built on top of on LSM-based storage systems, *TwoDB* and *DualDB*, to efficiently support high rates of dynamic subscriptions and publications (Section 5).
- We implemented our methods on LevelDB and conducted extensive experiments with various workloads with different subscription to publication ratios. For that, we extended LevelDB to handle lists of values per key (Section 4). Our experiments show that both approaches perform up to 1000% faster than baseline *RepQueries* (repetitive queries) and up to 3000% faster than a state-of-the-art pub/sub system Padres (Section 7).
- In addition to simple lookup subscriptions that return matching publications, we implemented methods to support subscription queries that join streaming publications/subscriptions with existing data/queries, which we refer as *self-joining subscriptions* (Section 6). We show that for such subscriptions *DualDB* performs better (up to 20%) compared to the *TwoDB* approach. Interestingly, *DualDB* performs 5% better than *TwoDB* even for simple match subscriptions (Section 7).
- To support hierarchical attributes, we propose a Dewey encoded representation of topics/cells and an efficient querying algorithm (Section 6). We show that our approach performs significantly better than reasonable baselines that handle multi-granular data (Section 7).
- We have published open-source version of our DualDB and TwoDB implementations on top of LevelDB [4].

The rest of the paper is organized as follows. Section 2 reviews the related work and background. Then, Section 3 presents the framework of the system, Finally, Section 8 concludes and presents future directions.

2 Related Work and Background

2.1 Related Work

2.1.1 Publish/Subscribe Systems

Most of the academic work on pub/sub mainly has studied how to efficiently route the message through the distributed pub/sub network. Instead, in this paper we propose an efficient data storage management system for pub/sub systems. There exist many variations of pub/sub systems supporting content or topic based subscription [14] [25]. Very few works studied the storage architecture of pub/sub systems. Padres [16] is a popular open-source pub/sub system, which allow continuous lookup queries. It supports subscriptions on future and past data, using a PostgreSQL database inside each broker [24]. Pub/Sub systems integrated in relational database systems have also been studied [8]. However, they do not scale with the number of subscriptions as we show in our experiments.

AsterixDB BAD [23] supports complex continuous queries, where queries are executed repetitively on top of the LSM-based AsterixDB database. Due to its repetitive nature, it does not provide instant notification. On the other hand, BAD supports a richer set of subscription query types than this work, where we require that subscriptions and publications match based on a key attribute condition. XML Based pub/sub systems also use a relational database for content based subscriptions [33]. Elaps [19] is a Location aware pub/sub system focusing on efficient processing of continuous moving range queries against dynamic event streams. Their work focuses on the query optimization rather than an efficient storage architecture. Our focus is on millions of dynamically arriving and expiring subscriptions and publications, which requires an efficient specialized storage framework. A preliminary version of our work was recently published as a short paper [30], which does not include the *TwoDB* approach, hierarchical attributes' support or self-joining subscriptions.

2.1.2 Triggers and Continuous Queries

Continuous queries on databases may be implemented using triggers [34] [31]. The TriggerMan project [20] proposes a method for implementing a scalable trigger system based on the assumption that some triggers may share a common structure. It uses a special selection predicate index and an in-memory trigger cache to achieve scalability. However, they are only able to handle a very small number of triggers on a table [12], whereas we want our continuous queries to scale to millions. Further, triggers have a relatively high creation and deletion cost, which makes them inappropriate for dynamically changing subscriptions.

NiagraCQ [12] proposed techniques to group continuous queries with similar structure, to share common computation. TelegraphCQ [10] is another system that handles streams of continuous queries over high-volume, highly-variable data streams. Complex continuous queries on data streams have also been studied [7] [27]. Kafka [17] is a popular stream processing platform. But our storage systems also support subscriptions on historic data where both subscriptions and publications dynamically expire, which data streams cannot support. Gemfire [1] is one of the few NoSQL database systems where continuous queries are supported. They maintain a separate in-memory processing framework, which maintains subscriber channels and communicates with the storage. InfluxDB [22] supports continuous queries on a time series database and thus its application is limited. Microsoft supports efficient spatial continuous queries on SQL Server [21]. However, these works assume that the queries fit in memory and are relatively static, that is, they do not scale to arbitrary numbers of queries nor to rapidly added and expiring queries. Further, these models generally only support "future-only" queries, that is, queries that only return future data items.

2.1.3 Join-Indices

Our work to combine two databases into one dual database is also related to join indices, such as Oracle's Bitmap join index [28], which creates a join index on attributes from two tables to facilitate faster joins. However, these are separate and specialized data structures that are generally expensive to maintain, in contrast to our proposed solution that does not add any redundant indexes and is efficient to maintain.

2.2 Background

2.2.1 LSM tree

An LSM tree generally consists of an in-memory component (a.k.a. Memtable) and several immutable on-disk components (a.k.a SSTables). Each component consists of several data files and each data file consists of several data blocks. As depicted in Figure 2, all writes go to in-memory component (C0) first and then flush into the first disk component once the in-memory data is over the size limit of C0, and so on. The on disk components normally increase in size as shown in Figure 2 from C1 to CL. A background process (compaction) periodically merges the smaller components to larger components as the data grows. Deletion on LSM is achieved by writing a tombstone of the deleted entry to C0, which will later propagate to the component that stores this entry.

LSM storage is highly optimized for writes as a write only needs to update the in-memory component C0. This append-only style means that an LSM tree could contain different versions of the same entry (different key-value pairs with the same key) in different components. A read (GET) on an LSM tree starts from C0 and then goes to C1, C2 and so on until the desired entry is found, which makes sure the newest (valid) version of an entry will be returned. Hence, reads are slower than

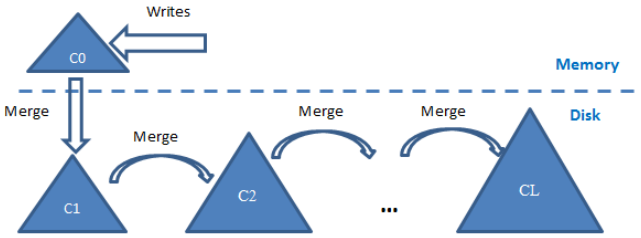
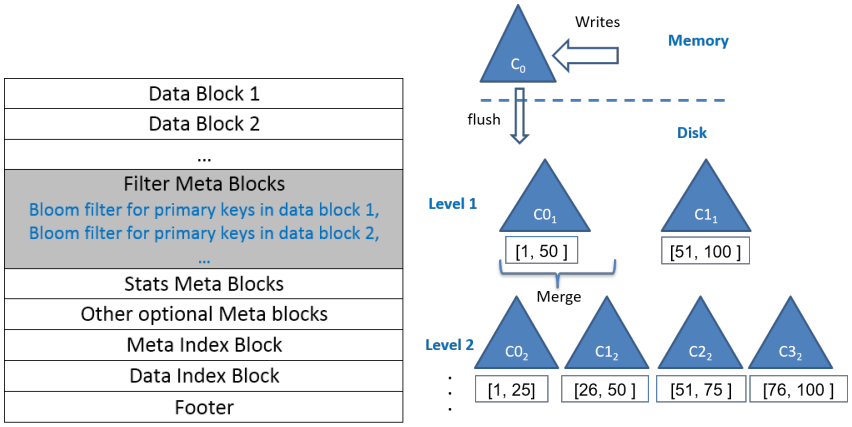


Fig. 2: LSM tree components.

writes. The older versions of an entry (either updated or deleted) are obsolete and will be discarded during the merge (compaction) process.

Because of LSM trees' good performance on handling high write throughput, they are commonly used in NoSQL databases such as BigTable [11], HBase [18], Cassandra [26], LevelDB [2], RocksDB [5] and AsterixDB [6].

2.2.2 LevelDB Storage Architecture



(a) LevelDB SSTable Format.

(b) LevelDB's leveled organization of SSTables.

Fig. 3: LevelDB Storage Architecture .

As we will implement our proposed approaches on top of LevelDB, we present some storage details of LevelDB as background for ease of understanding. The SSTables of LevelDB are organized into a sequence of levels where Level-$(i+1)$ is 10 times larger than level-i in LevelDB as shown in Figure 3b. Each level (component) consists of a set of disk files (SSTables), each having a size around 2 MBs. Here we can see that, the SSTables in the same level may not contain the same key (except level-0[1]). As described in the LSM storage, the lower level (smaller i) in LevelDB

[1] which is C1 in Figure 2 as LevelDB does not number the memory component.

will contain the more recent key-value pairs. Recent Cassandra versions support this level merging (compaction) option used in LevelDB.

Further, LevelDB partitions its SSTables into data blocks (normally tens of KB in size). The key-value pairs are stored in the *data blocks* sorted by their key. Meta blocks contain meta information of data blocks. Index blocks store the addresses of meta blocks and data blocks.

Compaction Policy: Similar to any LSM architecture, if any level L exceeds its limit, they compact it in a background thread. Level-0 files are just flushed memtables and that's why they have overlapping key-ranges. But for all other level, each file contains non-overlapping key-ranges. Their policy picks a file from level L and all overlapping files from the next level L+1 to merge. This selection for a particular level rotates through the key space of the files (i.e. Round-Robin). For example, in Figure 1, if $C0_1$ with key range [1,50] from level 1 is picked to merge with level 2, then it will merge with $C0_2$ and $C1_2$ as they contain overlapping key ranges ([1,25] and [26,50]) with $C0_1$. Compaction also drops overwritten/deleted values which ensures one unique key per level.

Need to extend LevelDB to support pub/sub: We summarize the reasons why we modified LevelDB vs. build an extra layer on top of it, to efficiently support pub/sub. First, we need to map one key (subscription topic) to many values (publications), and search based on the key. This is not possible in current LevelDB's implementation. We show that our modified storage (Section 4) can support efficient value lists with an intelligent lazy update strategy. Second, LevelDB does not support dynamic removal of expired pub/sub records. Specifically, we need to modify LevelDB to remove expired publications and subscriptions during background compaction. Third, LevelDB does not allow to store heterogeneous data (subscriptions and publications) in the same key space. Building a service layered on top of LevelDB would lead to much slower performance, especially due to the second reason above.

3 Framework

Table 1: Set of Operations for Pub/Sub Storage Framework

Operation	**Description**
SUBSCRIBE (*ID*, *Subscription-JSON*, *SelfJoinFlag*)	Store the subscription with *ID* as primary key; if T_{min} is in past ($T_S > T_{min}$), immediately return matching publications (publications with same *ID*), which are published in $[T_{min}, T_{max}]$ time interval ; if *SelfJoinFlag* is true, return matching list of subscriptions (see Section 6.1).
PUBLISH (*ID*, *Publication-JSON*, *SelfJoinFlag*)	store publication with *ID* as primary key; return subscribers who subscribed to this ID (i.e. topic/region); if *SelfJoinFlag* is true, also return matching publications .

To support pub/sub operations on top of a NoSQL data store, we define a basic API as shown in Table 1. To illustrate the functionality of this API, consider

the motivating application described in Section 1, where Subscriptions S (API call $SUBSCRIBE$ (ID, $Subscription\text{-}JSON$)) are generated by mobile users driving in their vehicles, who are interested to know about recent events, such as accidents or weather changes, close to their current location L_S. "Recent" may refer to events that happened in the time interval $[T_{\min} = T_S - 10 \text{ sec}, T_{\max} = T_S + 10 \text{ sec}]$, where T_S is the query (subscription) time. Note that the time interval associated with each subscription S can include both past and future time ranges. If the intervals only contained future times, no storage would be needed for the publications. Whenever there is a new publication (API call $PUBLISH$ (ID, $Publication\text{-}JSON$)), i.e. an event occurred, we look for all the stored subscriptions to notify them if the ID (cell-id for moving objects or topic-id for topic-based pub/sub systems) of the publication matches that of the subscription. Whenever there is a new subscription (API call $SUBSCRIBE$ (ID, $Subscription\text{-}JSON$)), we store the subscription in the database so that for future publications it can find matching subscriptions. Here, if the SUBSCRIBE operation queries for "past" ($T_S > T_{min}$) publications, it should immediately return all matching and valid publications published during time interval in the past [T_{min} , T_S] (or [T_{min} , T_{max}] , if $T_{min} < T_S$). Note that, for both operations, it should only return "valid" publications/subscriptions. Valid means it should return only those matching results which have not expired at current operation execution time ($T < Tmax$). We emphasize that subscription time intervals span both the past and the future, we need a storage framework to store the subscriptions as well as the publications.

ID is the key that joins subscriptions and publications. $Subscription\text{-}JSON$ and $Publication\text{-}JSON$ are the JSON-formatted values in our Key-Value store. They hold the attributes shown in Table 2. In this paper, we only consider topic-based pub/sub queries which match based on one attribute only (e.g. location, topic etc). Note that subscriptions may also specify additional conditions, such as keyword matching, for instance, return data close to me that contain the term "accident." Such conditions can be supported by a postprocessing (filtering) layer on top of the location-constrained or topic-constrained results. Another solution is to implement any of the existing secondary indexing techniques [29] on top of our proposed frameworks. These additional features are out of scope of this paper and we consider them as future works.

Table 2: Set of Attributes and Terminologies

ID	cell-id/topic-id/product-id
T_P, T_S	Execution time of a publication and subscription
T_R	Duration between two repetitive queries.
T_{min}, T_{max}	The time interval in a subscription query. T_{max} is expiry time for subscription and publication.
Sid, Pid	Subscription and publication Identifier
Uid	User Identifier (One user can submit multiple subscriptions)
$Desc$	Text Description field in publication
$Subscription\text{-}JSON$	Body of a subscription in JSON $\{T_{min}, T_{max}, T_S, Sid, Uid\}$.
$Publication\text{-}JSON$	Body of a publication in JSON $\{T_P, T_{max}, Pid, Desc\}$

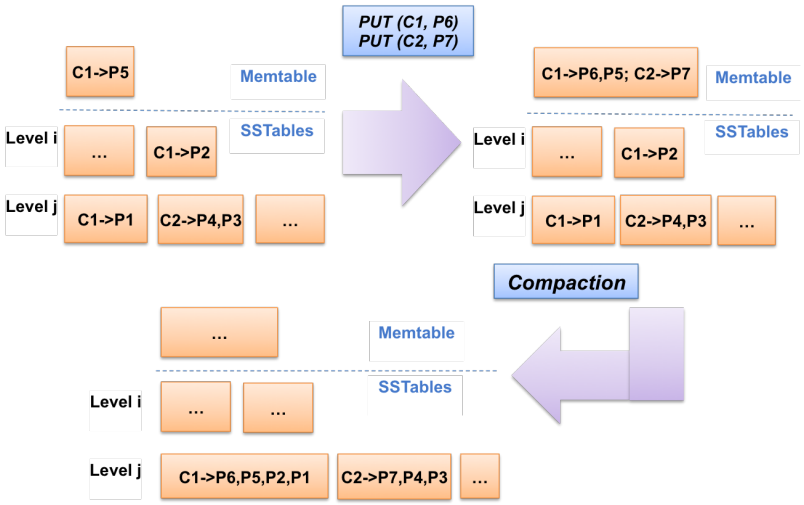


Fig. 4: Example of handling multiple writes (PUTs) with the same key, and a subsequent compaction.

4 Enable Value Lists in LevelDB

In this section we discuss how we modify LevelDB to handle lists of values instead of a single value for every key. For example, the list of events for a single cell id is stored as a value list with the cell id as key. LevelDB holds the LSM property shown in Figure 2 where data is first inserted in memtable which is flushed to disk components called SSTables later. SSTables like in Figure 3a hold records with unique key. We described how SSTables are organized into levels in LevelDB in Section 2.2.2. STables organized into levels in LevelDB and because of the way the compaction takes place, there can be one unique key in each level.

For our problem, we do not want uniqueness in primary key as we assume that for both *SUBSCRIBE* and *PUBLISH* operation, the location or cell-id is the primary key and we will match queries with events using this key. So, instead of one record associated with a key we will have a posting list of queries or events. So for each insertion with non-unique primary key/cell-id, we add the new record with the current list of records instead of dropping the old record. The naive way to do it will be an in-place update of the list by issuing a read on the current list, appending the new data to the list and writing back to the storage. Background compaction later will discard the old list. But here the main drawback is that each write becomes very expensive, as we need to read a large list. High write throughput is one of the fundamental features of NoSQL databases which we can not compromise. Also we may have a lot of invalid large lists throughout all the levels in SSTables, which will waste a lot space.

To solve this problem we implemented a lazy update strategy on the postings list. When a new Write/Put is issued on a record, we do not look for existing value lists associated with the same key/cell-id in the disk. We simply write them to the memtable. If there is a existing list in the memtable, we perform in-place update on the list in constant time. Memtable is flushed to SSTables when it is full and these SSTables are

compacted later to move to lower levels. We modify this compaction strategy appropriately, where instead of discarding old records, we merge lists associated with same key and hence eventually large lists are created in lower levels. Original compaction ensures that there can be at most one value list associated with a key in each level except level-0 files as SSTables in non-zero levels contain non-overlapping keys in LevelDB. Now we have fragmented list associated with a key throughout different levels. When we issue a read on key, instead of returning the record on upper level, it continues to search level by level to collect the fragmented lists. Example 1 illustrates how we maintain value lists in a lazy manner for each key in our key-valuelist storage.

Example 1 Assume the current state of a Publication database right after the following sequence of publish operations *PUT(C1,P1), ..., PUT(C1,P2), ..., PUT(C2,P3), PUT(C2,P4),..., PUT(C1,P5)*. Here *Ci* is the key and *Pi* is the value. Figure 4 depicts the state of the storage components (i.e. SSTables and Memtable) after these operations. Note that, the figure only highlights the records affected by these operations, but other records also exist in different components other than C1 and C2. We can see instead of key-value pairs we have value lists for keys C1 and C2, fragmented in components of different levels. Now, two new operations PUT(C1,P6), PUT(C2,P7) are issued. Figure 4 shows how it affects the current lists and how these lists are lazily updated during compaction. We first see the P6 is added to the list of C1 with an in-place update and a new record C2->P7 was created inside Memtable. Now after some compaction, these lists are moved to lower levels and we can see they are compacted and merged into larger lists (C1->P6,P5,P2,P1 and C2->P7,P4,P3). Here a read on C1 before the compaction would require to access the Memtable and SSTables in level i through level j. After compaction, a read requires to access an SSTable on level j only.

4.1 Cleaning up Invalid Records in Compaction

Our problem space explores to the big active data area where millions of subscriptions and publications can arrive and expire at a rapid pace. As they may have an attribute in their value (T_{max}), which denotes their expiration time. The storage management system should efficiently remove the expired items from its storage. Existing LevelDB storage does not support dynamic removal of expiry pub/sub records. Our proposed storage is built on top of LSM-tree which performs compaction in background. During compaction, we pick couple of SSTable files and compact them into a new set of files. During this time we sort and merge the list of publications or subscriptions associated with their ID, and check their expiration time against the current time to discard them if found invalid. This dynamic removal of expiry records during compaction does not allow LSM tree to grow indefinitely which makes our operations lot faster.

5 Proposed Approaches

We propose and implement several approaches to efficiently realize the API defined in Section 3, while providing instant response times to publications. We first present a reasonable baseline approach, *RepQueries*, which performs queries in a repetitive way and cannot provide instant responses. Next, we present our first novel instant-response approach (*TwoDB*), which maintains two databases for holding subscriptions and data records/publications. Our second approach (*DualDB*) maintains a single database holding both datasets. For all of the approaches we need to modify the standard key-value storage model to support lists of values per key (for details see Section 4).

5.1 Repetitive Queries Baseline (*RepQueries*)

RepQueries relies on a repetitive query approach and hence cannot guarantee instance delivery of a new publication. This approach is used in many pub-sub systems that use a broker management as a middle-ware between the database and the users. Here the user/broker is responsible for submitting the query repetitively to the publication database to find out the matching events/topics. Note that the user/broker is responsible to detect duplicate data if consecutive time ranges overlap. Also as the events are highly dynamic, some of them might expire in-between the repetitive queries and user can potentially face missing data.

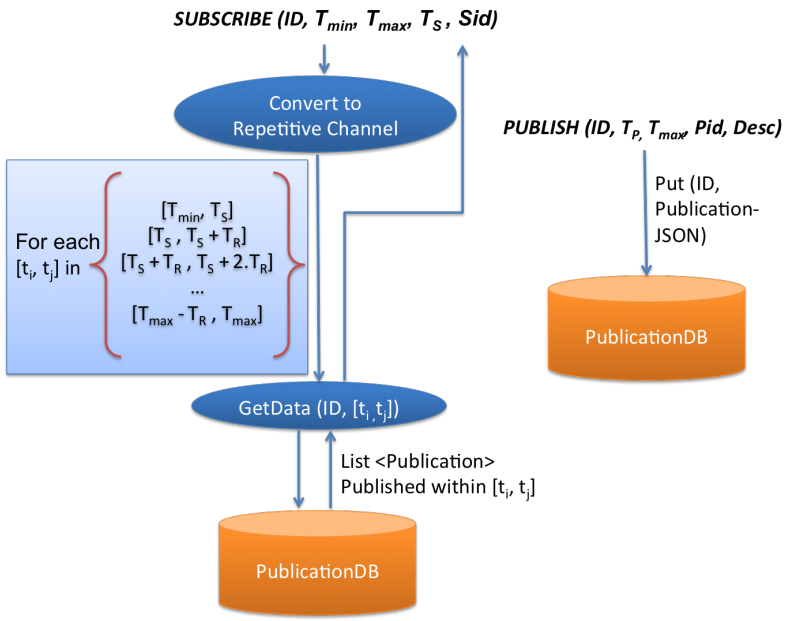
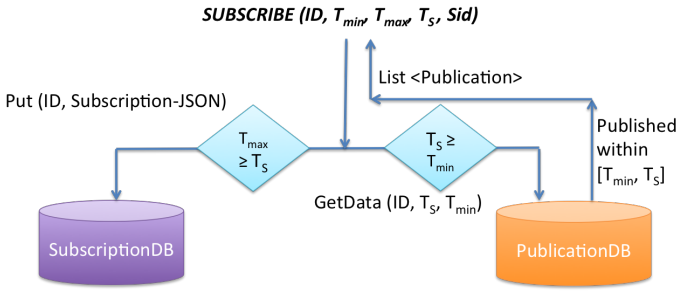


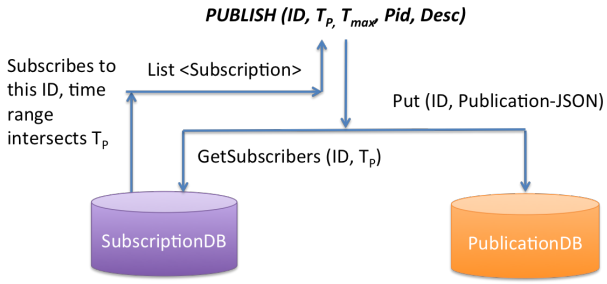
Fig. 5: **RepQueries Approach**: Processing of *SUBSCRIBE* and *PUBLISH* Operation

This approach only requires a database of publications. As periodic queries are issued from client application, and there is no need for instant response, we do not need to store the queries for future publications. A subscription is converted to a sequence of queries, one every T_R time units as shown in Figure 5. Specifically, it first issues a historic query on past data with time interval $[T_{min}, T_S]$, and then it repetitively issues *GetData* operation after every T_R time units. Here the PUBLISH operation does not issue any read, it only writes the new publication to the database. So PUBLISH will be very fast and SUBSCRIBE will be slow.

5.2 Two-Database Approach (TwoDB)



(a) Processing of new *SUBSCRIBE* Operation.



(b) Processing of new *PUBLISH* Operation.

Fig. 6: TwoDB Approach

As discussed in Section 2, previous works assume that the subscriptions are stored in memory, which is not realistic in our scenario. Figures 6a and 6b show the data flow for a new subscription (*SUBSCRIBE*) and a new publication of events/topics (*PUBLISH*), respectively, for a two-database algorithm, where separate databases (column families in Cassandra's terminology) are maintained for, respectively, the queries and the published events/topics.

For every new continuous query, we must, in parallel, insert to the subscription database *SubscriptionDB* and query the data storage *PublicationDB* for matching events/topics. The former is needed when the $T_{\max}$ of the subscription is greater than current query time T_S while the latter is only needed if the $T_{\min}$ of the query is less than T_S. The query to the publication database $GetData\,(cell-id, T_S, T_{min})$ returns the list of events which is published within time interval $[T_{min}, T_S]$ and are valid (i.e.did not expire) on that time. The procedure of *GetData* operation is presented in Algorithm 1. Since the postings list could be scattered in different levels, *GetData* needs to merge them to get a complete list. For this, it checks the Memtable and then the SSTables, and moves down in the storage hierarchy one level at a time.

Algorithm 1 GetData(ID, T_S, T_{min}) operation

```
1: ListPB ← ∅
   // Starts from Memtable (C0) and then moves to SSTable, C1 is LevelDB's level-0 SSTable.
2: for j from 0 to L do
3:     if ID is not in Cj then
4:         NEXT
5:     List of Publications P ← the value of key ID in Cj
6:     for Publication in P do
7:         T ← Publication(T_P)
8:         T_exp ← Publication(T_max)
9:         if T < T_S and T > T_min and T < T_exp then
10:            ListPB.add(Publication)
11:        if T < T_min then
12:            return ListPB
13: return ListPB
```

For each new publication of events/topics, we must, in parallel, check if an active subscription query matches it, and also insert it into the publication database. The matching function $GetSubscribers\,(ID, T_P)$ returns a list of subscribers who subscribed to some events/topics on same ID and the query is still valid and if the publication time T_P is within time interval $[T_{min}, T_{max}]$ of the query. The procedure of *GetSubscribers* operation presented in Algorithm 2 is similar to *GetData* operation moving down level by level in storage to look for matching subscriptions.

Algorithm 2 GetSubscribers(ID, T_P) operation

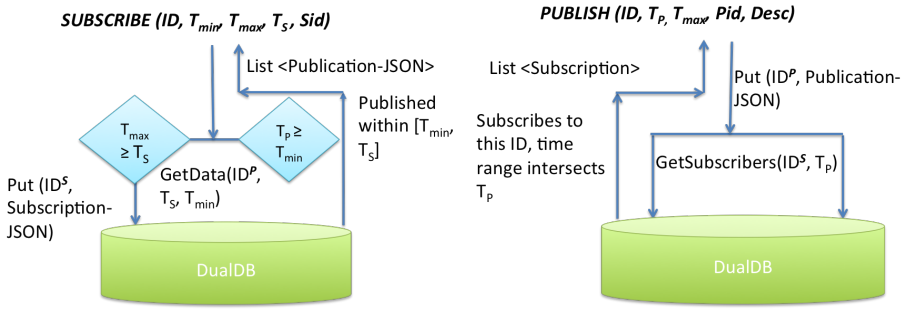
```
1: ListSB ← ∅
   // Starts from Memtable (C0) and then moves to SSTable, C1 is LevelDB's level-0 SSTable.
2: for j from 0 to L do
3:     if ID is not in Cj then
4:         NEXT
5:     List of Subscriptions S ← the value of key ID in Cj
6:     for Subscription in S do
7:         T_exp ← Subscription(T_max)
8:         T_min ← Subscription(T_min)
9:         if T_P < T_exp and T_P > T_min then
10:            ListSB.add(Subscription)
11: return ListSB
```

5.2.1 Pruning search in SSTables

To answer a subscription that has a T_{min} in the past, we take advantage of an important characteristic of level-based LSM systems. In LevelDB, each level is ordered by time and older records go into lower levels. In our storage, each level can contain at most one list associated with a key. That means the fragmented lists associated with a key in different levels from top to bottom are ordered by time. Here, the publications in each lists are also ordered by execution time T_P. So for each SUBSCRIBE operation, when we perform *GetData* operation to look for all matching publications from past (T_{min}), we check for associated lists going level by level. For each matching list, we parse the list to check each publication record one by one. If we find one of them is older than T_{min}, we can safely assume that we neither need to go any further down the level nor further right to this current list, and hence we stop the search. Lines 11-12 in algorithm 1 show this optimization.

5.3 Single Database Approach (DualDB)

A key observation is that *if continuous subscription queries and data publications could be stored in the same key space, then a single database (in LevelDB terminology) could store both of them*. Then query and data insertions would only need to access a single database, which could reduce the number of disk accesses and more importantly improve the caching efficiency. Further, the compaction cost might be decreased, as we only have to compact a single database.



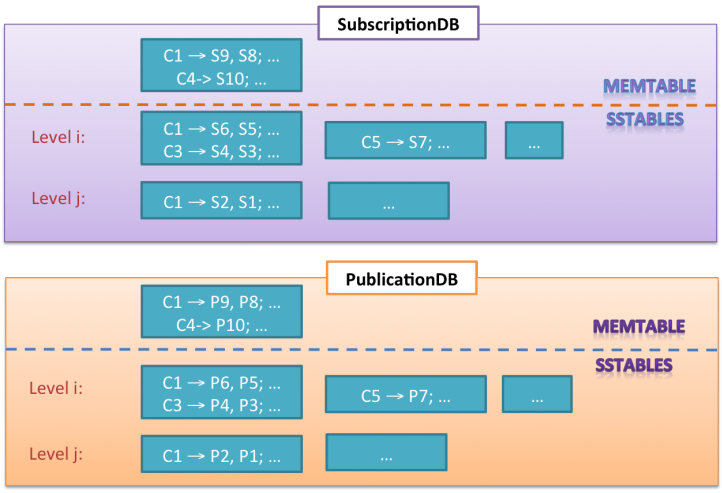
(a) Processing of a *SUBSCRIBE* Operation.

(b) Processing of a *PUBLISH* Operation.

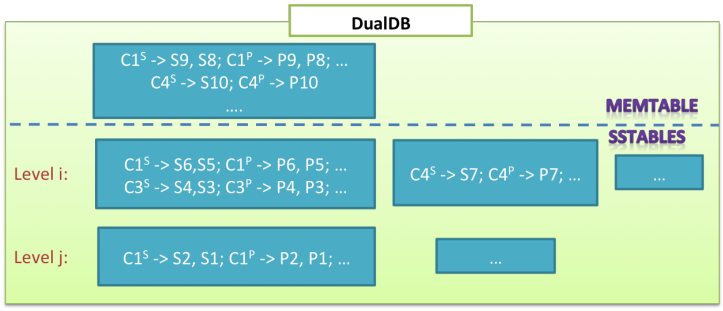
Fig. 7: DualDB Approach

We propose to study the properties and performance of using a single database, which we call a *dual database (DualDB)*. The key idea is that the key-value data organization must be modified to accommodate a list of subscription and publication items in the value. That is, for a given ID (cell-id or topic-id) in our example, both the list of subscriptions and publications will be stored in the posting list of this

ID. As LevelDB does not allow same key inside a component, we need to change the storage architecture to allow the disk and memory components holding at most two lists associated with the same key/cell-id. We assign a bit with the primary key (ID^S, ID^P), which will state whether it's a list of publications or subscriptions. Figures 7a and 7b show, respectively, how new query and new publication of events are processed in the proposed *DualDB*.



(a) Two-Database Approach



(b) Single Database Approach

Fig. 8: Snapshot of storage components for our LevelDB-style LSM database for pub/sub System.

We can see that the key difference with the TwoDB approach is using single database against two and all the operations *Put*, *GetData* and *GetSubscribers* are performed with same *ID* with a assigned flag stating whether its a query or data. Here both *GetData* and *GetSubscribers* procedure follows the same algorithms 1 and 2 of TwoDB approach respectively. The optimization techniques for lookups described in

Section 5.2.1 can also be applied for *DualDB* in similar fashion. *DualDB*, similarly to *TwoDB*, cleans up the invalid records during compaction as described in Section 4.1.

As we are combining two databases into one database containing two lists, the storage components are changed accordingly to hold any such two heterogeneous data. Figures 8a and 8b show, respectively, snapshots of the entries in the corresponding databases for the Two-Database approach and the DualDB approach. In Figure 8a, we can see that the TwoDB approach holds posting lists of subscriptions (S1,S2,...) and publications (P2,P1,...) in two different databases and each list associated with a ID (C1,C2 etc.) is fragmented throughout different levels. Figure 8b shows the DualDB is holding lists of subscriptions and publications in same components with same key ID which holds a flag ($C1^S$,$C1^P$) that distinguishes between them. As in each component the records are sorted by key, these two lists will always be co-located with each other. This should allow improvements in both LevelDB memory cache and the operating system page cache. These figures assume a leveled storage architecture (as in LevelDB and Cassandra); a stack-based architecture can be handled similarly.

A broker in a pub/sub system can utilize these SUBSCRUBE and PUBLISH APIs as continuous queries if the storage layer is built on top of *TwoDB* or *DualDB*. Here compared to the repetitive approach, both of our approaches work on continuous queries and can guarantee that there is no data loss.

6 Extend to Complex Subscriptions

6.1 Self-Joining Subscriptions

As discussed in Section 5, our *TwoDB* and *DualDB* approaches can efficiently process simple publications and subscriptions. However, some scenarios require publications to also access other publications, in addition to querying the subscriptions (the same holds for subscriptions). We call these *self-joining publications (subscriptions)*.
Motivating Examples. Consider mobile users moving and subscribing to events on a region or cell. Imagine the scenario where a mobile user enters cell c1, then c2 and then c1 again, all within 10 seconds, and all subscriptions have a time interval of $\pm$10 sec. A desirable property for many applications is that data already returned when the user was in c1 the first time should not be returned again during the second entry. For that, the system must efficiently identify what data has already been returned to a user. This is a classic use-case for using self-join subscriptions where for a new subscription, we need to access not only the list of past matching publications, but also access the list of matching past subscriptions from the user in order to filter these results. As another application, consider a user who subscribes to a sensor temperature (or to a stock price) and wants to be notified if there has been a sudden change/drop in temperature/price. In this case, for a new publication, we need to traverse both the publications list to check past temperatures/prices in this location/stock, and the subscriptions lists to know who subscribes here.

That is, for these operations, a subscription (publication) must join with past subscriptions (publications), hence the name *self-joining* subscriptions. Our existing stor-

age framework gives us flexibility to support such subscriptions efficiently. For both our approaches, if $SelfJoinFlag$ is set we assume that it is a self-joining operation.

For a self-join subscription, we write our subscription and/or publication to the storage and retrieve a list of events and a list of queries from storage by issuing a $GetDataANDSubscribers(ID, T_{P/S}, T_{min})$ operation. Here $T_{P/S}$ means execution time of either publication or subscription.

Algorithm 3 GetDataANDSubscribers $(ID, T_{P/S}, T_{min})$ operation on $TwoDB$

1: $ListPB \leftarrow this.PublicationDB.GetData(ID, T_{P/S})$ // [Algorithm 1]
2: $ListSB \leftarrow this.SubscriptionDB.GetSubscribers(ID, T_{P/S}, T_{min})$ // [Algorithm 2]
3: **return** $ListPB$, $ListSB$

Algorithm 4 GetDataANDSubscribers$(ID, T_{P/S}, T_{min})$ operation on $DualDB$

1: $ListPB \leftarrow \emptyset$
2: $ID^P \leftarrow Setflag(ID, E)$
3: $ListSB \leftarrow \emptyset$)
4: $ID^S \leftarrow Setflag(ID, Q)$
// Starts from Memtable (C0) and then moves to SSTable, C1 is LevelDB's level-0 SSTable.
5: **for** j from 0 to L **do**
6: **if** ID^P is not in Cj and ID^S not in Cj **then**
7: NEXT
8: $ListSB \leftarrow$ // Apply Line 5-10 in Algorithm 2 for key ID^S .
9: $ListPB \leftarrow$ // Apply Line 5-12 in Algorithm 1 for key ID^P .
10: **return** $ListPB$, $ListSB$

In the TwoDB approach, the implementation for self-join subscription is straightforward: we simply perform an extra GetSubscribers $(ID, T_{P/S})$ operation (Algorithm 2) on PublicationDB (along with $GetData$ operation) to return the list of active matching subscribers for SUBSCRIBE operation and perform an extra GetData $(ID, T_{P/S}, T_{min})$ operation ((Algorithm 1)) on PublicationDB (along with $GetSubscribers$ operation) to return the list of valid publications for PUBLISH operation.

In DualDB, both the lists of publications and subscriptions associated with the same ID are stored in the same disk block next to each other. We convert the $GetSubscribers$ and $GetData$ operations performing on same Dual database into a single operation, which means we can get the both the lists of publications and subscriptions on single operation. We first convert the key ID to ID^P and ID^S by setting appropriate bits. Then we search in our DualDB starting from Memtable and moving to SSTables level by level. If we find a match for either ID^P or ID^S , we fetch the resulting block form the storage components which might contain both the lists. We extract both the lists if they are available in that particular fetched block. Here we can see that one disk I/O might sufficient against two if the list of publications and subscriptions are co-located inside same storage file block.

6.1.1 Cost Analysis (TwoDB Vs DualDB Vs RepQueries)

Table 3 summarizes the number of disk accesses for each PUBLISH/SUBSCRIBE operation for simple and self-joining subscriptions.

For simple subscriptions, only one I/O is needed to write the subscription and the publication in SUBSCRIBE and PUBLISH operation respectively. For both the operation, we issue a read (GetData/GetSubscribers operation) on the database which may involve up to L (number of levels) disk accesses (because in worst case the postings list is scattered across all levels). Here, at most one disk access is required for level except level-0. As level-0 contains files with overlapping key ranges, we need to traverse all files inside level-0. Let us assume l_0 is the number of level-0 files. As both TwoDB and DualDB approach issue one write and one read for their each operation, the total cost becomes $(L+l_0)+1$ disk accesses for both of them. *RepQueries* performs $\frac{T_{max}-T_S}{T_R}$ repetitive queries on publication database instead of a single subscription. But *RepQueries* publish operation only writes the publication record to the database.

For Self Joining subscriptions, similarly one write is needed for each operation. But for TwoDB approach, we need to submit two reads in two databases of publication and subscriptions (one in PublicationDB and one in SubscriptionsDB). This makes the total cost as $2*(L+l_0)+1$ disk accesses for each PUBLISH or SUBSCRIBE operation. But DualDB approach (Algorithm 4) only issues one read in dual database and the total cost becomes $(L+l_0)+1$ disk accesses in the worst case. Note

Table 3: Number of disk accesses for PUBLISH/SUBSCRIBE operation for simple vs self-joining subscriptions.

Approach:Operation	**Simple Subscription**	**Self-Join Subscription**
TwoDB : *SUB*	$(L+l_0)+1$	$2\cdot(L+l_0)+1$
DualDB : *SUB*	$(L+l_0)+1$	$(L+l_0)+1$
RepQueries : *SUB*	$\frac{T_{max}-T_S}{T_R}\cdot(L+l_0)$,	Not Supported
TwoDB : *PUB*	$(L+l_0)+1$	$2\cdot(L+l_0)+1$
DualDB : *PUB*	$(L+l_0)+1$	$(L+l_0)+1$
RepQueries : *PUB*	1	Not Supported

that we assume there is only one disk access per write operation. But in practice, LevelDB suffers from high write amplification factor where same data is written multiple times when moved from one level to another. This factor is excluded from our calculation.

6.2 Support for Hierarchical Attributes

In this section, we introduce an optimization in our storage system to efficiently support multi-granular data, which can be defined by a hierarchical model. For example, we may have a spatial hierarchy (e.g. Country->State->County->City->Block),

where each publication or subscription may select spatial cells of different granularity. Similarly, topics from a topic hierarchy can be selected. We propose and compare two approaches to support such hierarchical attributes in a pub/sub storage system. Note that the choice of an approach is orthogonal to the choice of a storage architecture from the ones described in Section 5.

6.2.1 Fixed Granularity Approach

In this approach, both subscriptions and publications use the lowest level keys in the hierarchy. For example, if a subscription specifies a city which has 100 blocks, then we insert 100 copies of the subscription. Similarly we handle publications. A disadvantage is that multiple entries are needed for each subscription or publication. The key advantage is the simplicity, as this can be implemented as a preprocessing step on top of any storage framework.

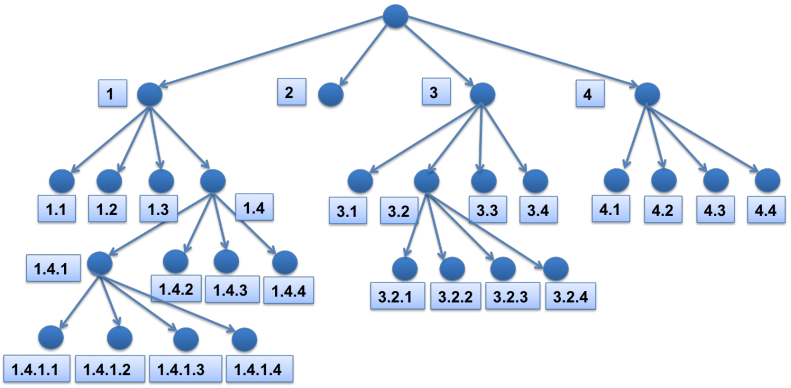
6.2.2 Variable Granularity Approach

In this approach, we only have one entry per subscription or publication, and the storage framework manages the matchings. We propose to use Dewey encoding [32] (incremented integer numbers separated by dots) to represent each entity in the ontology, and this Dewey id becomes the key (ID) of the publication or subscription. Let us assume a hierarchy is defined by a quad tree. Figure 9a illustrates how each node in different depth of the quad tree can be represented by a Dewey encoding. Figure 9b illustrates how we partition the space using that quad tree. During a *SUBSCRIBE*/*PUBLSIH* operation, we write the entry (publication/subscription) to the database and issue a range query from its Dewey to the Dewey of its next sibling.

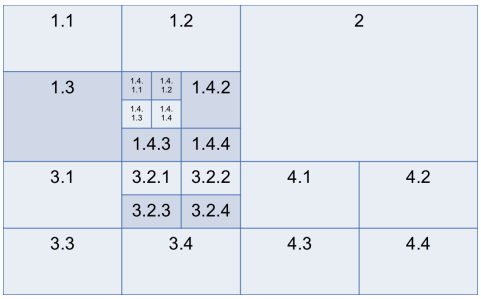
For example given the quad tree partition in Figure 9, let a publication/subscription cover area (1.4) (or represent topic (1.4) if we use topics instead of locations as ID). The fixed granularity approach will issue PUBLISH/SUBSCRIBE operation for each areas/topics in the leaf level of the subtree of (1.4) [(1.4.1), (1.4.1.1), (1.4.1.2),(1.4.1.3), (1.4.1.4), (1.4.2), (1.4.3), (1.4.4)]. But in our optimized variable granularity approach, we will issue one write for this topic/area (1.4) and issue a range query from (1.4) to (1.5).

Next, we discuss how a range query based on Dewey IDs is efficiently supported, by modifying the approaches in Algorithm 1 and 2. We iterate from upper level to lower level, and for each level when we find the file that contains the start key, retrieve the value list associated with that key, and then instead of returning from there we continue iterating through that file (or next file if necessary) until the end key. As we know that the data in our storage files (i.e.SSTables) are lexicographically sorted by their keys (i.e. Dewey numbers) and partitioned into fixed size blocks. So given a Dewey Number representing an internal node, all the nodes descended from that node by pre-order traversal should be stored. That makes this range query very efficient.

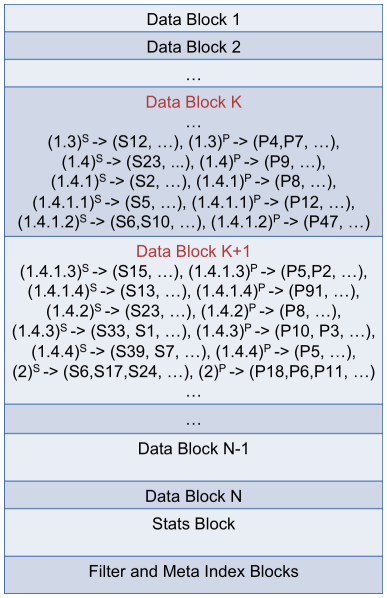
Figure 9c represents a snapshot of an SSTable in our DualDB, where the spatial hierarchy is represented by a quad tree in Figure 9a and 9b. Here we can see that how



(a) A Quad Tree and Dewey Encoding for each Node



(b) Partitioning Space using Quad Tree and Dewey Representation in Leaf Nodes



(c) Snapshot of an SSTable in DualDB containing lists in order of Dewey ID of the Key

Fig. 9: Variable Granularity Approach by Using Dewey Encoding

the list of publications or subscriptions in SSTable are sorted by its key lexicographically. If we issue a range query from (1,4) to (1,5), our algorithm will look for the file which contains (1,4) in each level, retrieve the disk block K of that SSTable and retrieve the list of subscriptions/publications associated with (1,4). Then it will retrieve the next records of that disk block subsequently [(1,4,1),(1,4,1,1),...] until it reaches the node (2). It might retrieve subsequent disk blocks in order to reach (1,5)/(2). In this figure, the last record of the block K is (1.4.1.2) and we need to retrieve block $(k+1)$. Note that similarly to our simple point lookup algorithm in Section 5.2.1, we check the time predicates of all the publications/subscriptions in each list for validity, and return early if pruning can be utilized.

Algorithm 5 GetDataInRange ($StartID^P, EndID^P, T_P, T_{min}$)

```
1: ListPB ← ∅
   // Starts from Memtable (C0) and then moves to SSTable organized into L Levels. C1 is LevelDB's level-0 SSTable.
2: for j from 0 to L do
3:     ListSSTs ← getOverlappingSSTablesInsideLevel(j, StartID^P)
4:     for f in ListSSTs do
5:         Iterator ← getIterator(Cf)
6:         Iterator ← Iterator.Seek(StartID^P)
7:         while Iterator is Valid do
8:             if Iterator.Key.getFlag() is Publication then
9:                 List of Publications P ← Iterator.Value
10:                ListPB ← // Apply Line 6-9 with value of T_P and T_min in Algorithm 1 to Populate ListPB from P
11:            Iterator ← Iterator.Next()
12:            if Iterator.Key >= EndID^P then
13:                return ListPB
14: return ListPB
```

Algorithm 5 summarizes how a single lookup *GetData* operation (Algorithm 1) for simple subscription queries can be extended to support a range operation *GetDataInRange* based on Dewey representation. Algorithm 2, 3 and 4 are extended similarly to support range operation on hierarchical attributes for simple and self-joining subscriptions.

6.2.3 Cost Analysis (Fixed Vs Variable Granularity Approaches)

We derive in Section 6.1.1 that a single PUBLISH or SUBSCRIBE operation can cost $O(L) = L + l_0 + 1$ number of disk accesses for both simple and self join subscriptions in *DualDB*. If we define the hierarchy as quad tree and the maximum depth as d, the number of leaf nodes in the worse case will be 4^d. So the cost for one operation will be $4^d * O(L)$ as we issue one operation for all leaf nodes separately. But for the variable granularity approach, we issue one write and a range query. Each range query similarly can hit $L + l_0$ number of files in each level and for each file, we may retrieve C consecutive disk blocks. C can be computed by dividing the size of the result set by the block size. The cost for each operation will be $C * (L + l_0) + 1$ number of disk accesses.

Table 4: Number of Disk Accesses for Hierarchical Attributes

Approach	Approximate I/O Cost
Fixed Granularity	$4^d * O(L)$
Variable Granularity	$C * (L + l_0) + 1$

7 Experiments

7.1 Experimental Setup

We ran our experiments on a machine with the following configuration: Processor of AMD Phenom(tm) II X6 1055T and 8GB RAM, with Ubuntu version 15.04.

7.1.1 Data and Query Workload

We used the Twitter streaming API to collect 15 million geotagged tweets, taking 12 GB in JSON format, located within New York State. We use this dataset to generate our desired workloads for the experiment. We ran experiments using different Subscription-to-Publication ratios. As there are too many parameters, we set the time intervals for each query and expiration time of each event to a constant value: all subscriptions have T_{min} as 10 seconds behind current time T_S and T_{max} as 10 seconds ahead of T_S, and all the publications have expiration time (T_{max}) of 20s ahead of current time T_P. For baseline *RepQueries*, we convert each SUBSCRIBE operation into 10 repetitive queries, each is executed after every $T_R = 1$ second interval.

There are several public workloads for key-value store databases available online such as YCSB [13]. However, to simulate a highly dynamic publish/subscribe model, we need to generate a mix of dynamically expiring subscriptions and publications. We decided to write our own script to generate different workloads from the real twitter dataset. Our workload generator considers each tweet as either a subscription or a publication depending on the Subscription-to-Publication ratio.

As all our tweets were collected within New York State, we use the bounding box rectangle around New York State and partition it to generate $500 * 500$ uniform sized cells each having a unique identifier cell-id. We map the Geo-location of the tweet to appropriate cell-id and use this cell-id as a primary key for input. If the tweet is an event (publication), the text is considered as event description. Tweet ID is used as either subscriber ID or publication ID. The time intervals are set according to the execution time of that particular operation as discussed above. In our dataset, as we have about 0.25 million cells and 15.3 million tweets, average number of tweets per cell is about 60 and as described, these tweets are converted into events and subscriptions.

7.1.2 Padres Baseline

In addition to the repetitive queries baseline (RepQueries) described in Section 5.1, we also compare to a popular Pub/Sub system, Padres. We understand that Padres is a content-based pub/sub and it also has a client-server architecture, which may incur additional overhead in delivering a subscription result, but we show that the performance difference is quite dramatic, which would dominate such overhead. First, we have to express our problem setting using Padres' model. For that, we convert our workload into Padres subscription and publication operations. Padres supports *historic subscriptions* on past queries. So, each subscriptions in our dataset is equivalent to one historic subscription and one regular subscription in Padres. Each event

publication is also converted to a publish operation in Padres. We installed Padres locally containing one broker and two clients connecting to the broker. One client is subscribing queries and the other client is publishing events. For clarification, let's illustrate the dataset conversion with an example. Suppose at T_S, a subscription query with cid as cell-id and interval $[T_{min}, T_{max}]$ is issued. We then convert it to a composite subscription (Expression 1) and a regular subscription (Expression 2).

$$CS[class, eq, historic], [subclass, eq, events], \\ [T, <, T_S], ..., [cellid, eq, cid] \& [T, >, T_{min}], ..., [cellid, eq, cid] \quad (1)$$

$$S[class, eq, events], [T, <, T_{max}], ..., [cellid, eq, cid] \quad (2)$$

A new generated event publication at T_P with cid as cell-id and T_{max} as expiration time will be converted to the following publication operation (Expression 3).

$$P[class, events], [T, T_P], [description, any], [cellid, cid] \quad (3)$$

7.2 Experimental Results

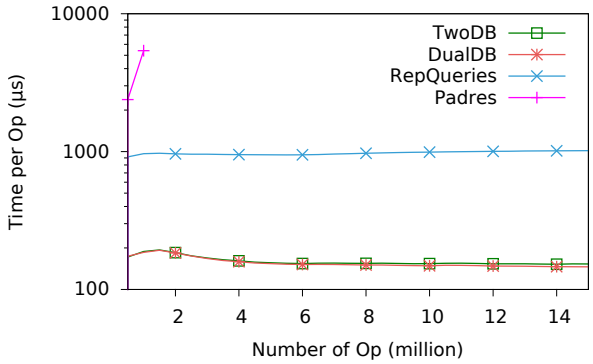
We conduct our experiments on workloads for both simple matching subscription and more advanced self-joining and multi granular subscriptions for different subscription-to-publication ratios, which represent different use cases.

7.2.1 Simple Subscriptions

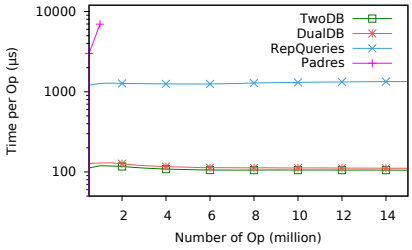
Figures 10 and 11 show the overall, SUBSCRIBE and PUBLISH performance of all systems subscription heavy ($\frac{Subscription}{Publication} = 3$) and publication heavy ($\frac{Subscription}{Publication} = \frac{1}{3}$) workloads, respectively. For example, our moving subscriber application is a use-case for subscription-heavy workloads whereas following Twitter accounts is a use-case of publication-heavy workloads. In all figures, we record the performance once per million operations. We display the cumulative total time taken for both PUBLISH and SUBSCRIBE operation separately and also collectively and calculate average time per operation in every million operations.

Figures 10 and 11 show that if the system relies on repetitive queries instead of instant response queries, it can not scale to millions of operations. As *RepQueries* does not issue any read after each publication, and only issues a write to a single database, PUBLISH has very good performance as expected. But SUBSCRIBE has bad performance as *RepQueries*. The overall performance is far worse than our proposed instant-response variants. Both *TwoDB* and *DualDB* approaches have about 1000% better performance than *RepQueries* for subscription heavy workload, and 300% better for publication heavy workload.

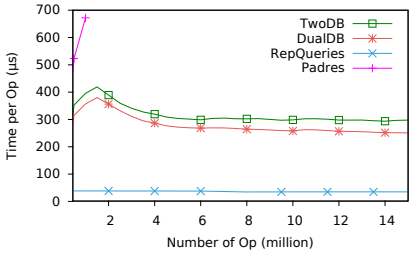
According to the theoretical analysis in Section 6.1.1, both *TwoDB* and *DualDB* approaches cost the same number of disk accesses for each PUBLISH and SUBSCRIBE in the worst case. This is confirmed experimentally in Figure 11, where we find that they have identical performance for publication heavy workload.



(a) Overall performance



(b) SUBSCRIBE performance

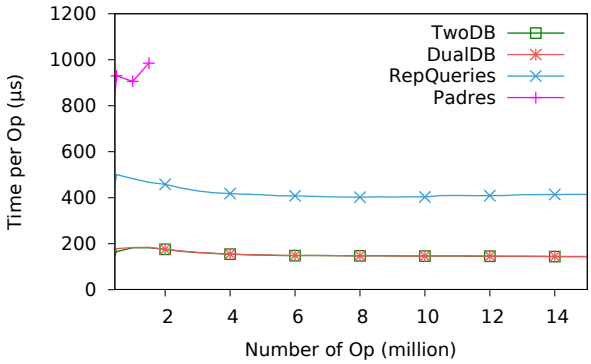


(c) PUBLISH performance

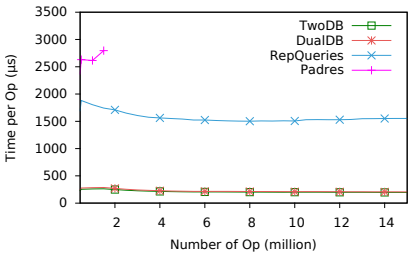
Fig. 10: Performance of different storage variants for simple subscription queries on subscription heavy workload

Figure 10 shows the results for subscription heavy workload, where *DualDB* is better than *TwoDB* by about 5%. This behavior is expected in *DualDB*, although it does not perform any extra disk access than *TwoDB* theoretically. The reason is that *DualDB* holds both query and data associated with a key inside the same disk block or in a neighbor disk block, which leads to better cache performance for both OS page cache and LevelDB block cache.

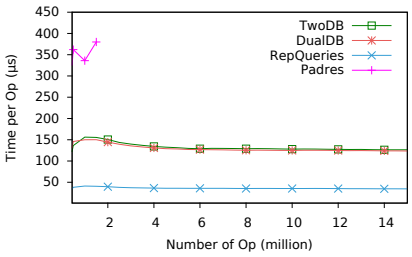
We see in Figures 10 and 11 that Padres performs poorly not only compared to our two approaches, but also to the repetitive baseline. Also, in our experiments Padres is not able to cope with the increasing number of subscriptions, and runs out of memory very quickly (even before 2 million operations) and the system crashes. This is because it relies on an in-memory data structure to manage subscriptions and fails to cope with a very large number of queries. Note that we allocated maximum memory for Padres (i.e. 8GB) and it still runs out of memory. Here we can see that even if we have sufficient memory to support small number of subscriptions, the use of traditional SQL-like database perform much worse (up to 300%) than even our baseline *RepQueries* approach.



(a) Overall performance



(b) SUBSCRIBE performance



(c) PUBLISH performance

Fig. 11: Performance of different storage variants for simple subscription queries on publication heavy workload

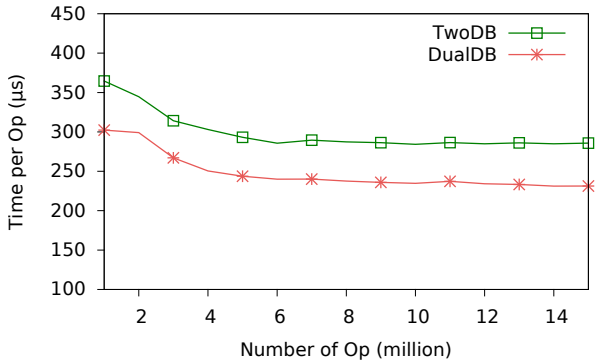
7.2.2 Self-Joining Subscriptions

Similar to the simple matching subscription, we observe the performance for complex self-joining subscriptions. Figures 12 and 13 show the overall, SUBSCRIBE and PUBLISH performance of *DualDB* and *TwoDB* approaches on both subscription and publication heavy workloads, respectively. Here we do not consider baseline *RepQueries* approach, which relies on repetitive queries on Events database, because it does not store the subscriptions in a database and hence it cannot perform self-joining queries.

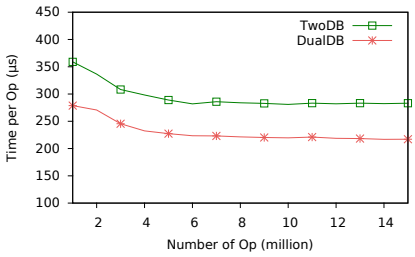
Figure 12 shows that *DualDB* is overall 20% (About 6% on PUBLISH and About 25% on SUBSCRIBE) faster than *TwoDB* for subscription heavy workload. We see similar performance gain of 12% overall improvement for *DualDB* over *TwoDB* in publication heavy workload. Note that here PUBLISH performs much better than subscription heavy workload which is desired in a publication heavy workload.

7.2.3 Subscriptions for Multi Granular Data

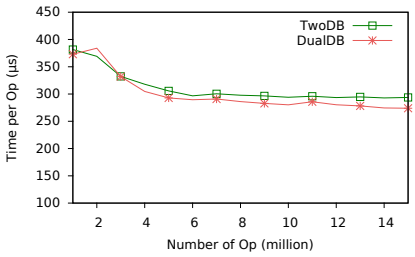
To simulate the environment for multi granular attributes, we partitioned the space by a balanced quad tree of depth 5 instead of a uniform grid. Here each cell represents



(a) Overall performance



(b) SUBSCRIBE performance



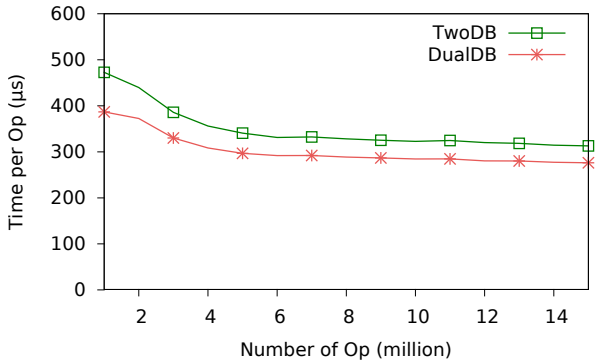
(c) PUBLISH performance

Fig. 12: Performance of different storage variants for Self-joining subscriptions on subscription heavy workload

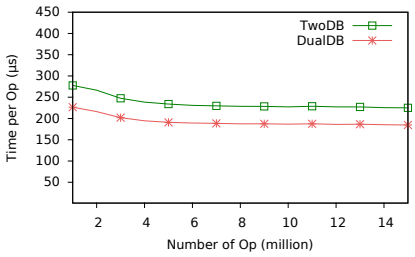
a node in the tree and each cell-id is encoded by Dewey Numbering (described in Section 6.2 and Figure 9). We set subscription-to-publication ratio as 1. This workload generates a mix of subscriptions and publications from different levels. We argue that the smaller cells will likely be accessed more than larger cells up in the hierarchy. For that, we design the workload generator in such way that a cell in level i has probability proportional to $\frac{P}{2^{d-i}}$ to be selected, where d is the maximum depth of the tree. In parallel, we also generated the workload for the fixed granularity approach by translating each operation into a multi operation (one operation for all the lowest level cells).

Figure 14 shows the overall, SUBSCRIBE and PUBLISH performance of *DualDB* for this workload of variable granularity against the fixed granularity approach. We see that our intelligent range query based variable granularity approach performs about 300% better for both PUBLISH and SUBSCRIBE operation. Figure 16 shows the performance for both approaches in different levels. It shows the average time per operation after performing 15 million operation. We see that our variable granularity approach performs even better for upper levels in the hierarchy. These results justify the cost analysis in Section 6.2.3.

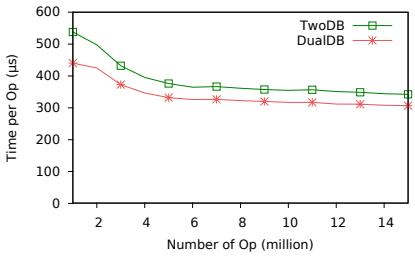
In the above experiment, we assume that a subscription or publication must pick one of the existing geospatial regions of the quad tree. Next, we show how arbitrary



(a) Overall performance



(b) SUBSCRIBE performance



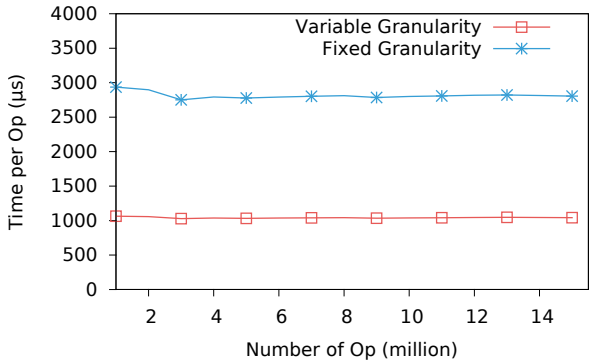
(c) PUBLISH performance

Fig. 13: Performance of different storage variants for Self-joining subscriptions on publication heavy workload

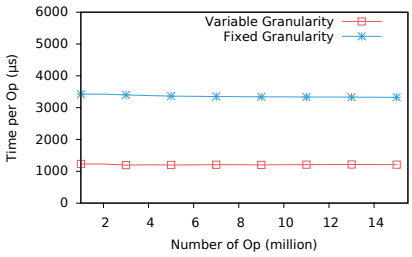
spatial regions, specifically circles can be supported, represented by a geo-location point and a radius. We generated a workload from our Twitter dataset for this application, where we select a radius for each operation from a set of five radii ($4*R$, $2*R$, R, $R/2$, $R/4$). Here, R is the length of the smallest cell in the quad tree. For the variable granularity approach, each operation's geo-location and a randomly selected radius (from the pre-defined set) is mapped to the cell in our quad tree whose area fully contains the circle region. For fixed granularity approach, this circle is mapped to all the cells in the lowest level whose rectangular region intersect with this circle. We perform one operation for all of these cells separately. Figure 15 shows that our variable granularity approach performs about 200% better for both PUBLISH and SUBSCRIBE operations. Figure 17 shows the performance for both approaches for different radii. We see that the performance gain of our variable granularity approach becomes greater for larger radii.

8 Conclusions and Future Work

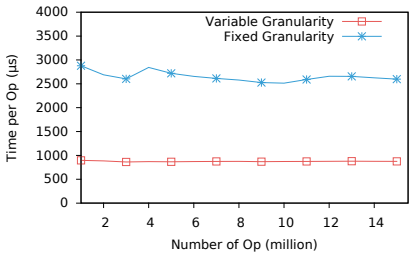
In this paper we present efficient storage and indexing approaches to achieve high throughput publish/subscribe on LSM-based databases where both the number of



(a) Overall performance



(b) SUBSCRIBE performance



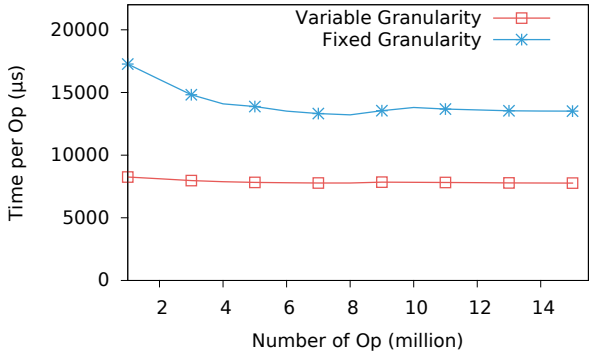
(c) PUBLISH performance

Fig. 14: Performance of Pub/Sub in DualDB for high granular data in Different levels of hierarchy for whole workload

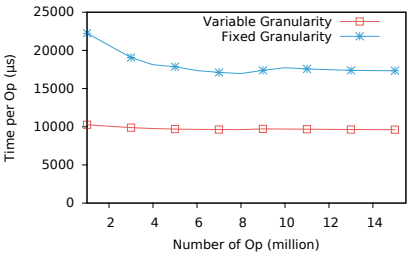
subscriptions and publications are massive in scale and one or both of them can arrive and expire with time. Our approaches support instant notifications. We also consider a baseline approach that relies on repetitive queries. We also show how hierarchical attributes in multi granular data can be supported efficiently by a Dewey encoded representation.

We implement these storage frameworks on top of the popular LSM-based LevelDB system and conduct extensive experiments using real datasets. In contrast to our *TwoDB* approach, *DualDB* approach is harder (in terms of lines of code) to implement on an existing NoSQL storage, but it offers better performance and more flexibility. The experimental results show that our proposed approaches outperform the state-of-the-art *Padres* pub-sub system (by up to 3000%) and also outperform the repetitive baseline *RepQueries* (by up to 1000%). We also show that our *DualDB* approach outperforms the *TwoDB* approach for simple subscriptions by a small margin (up to 5%) and for self-joining complex subscriptions by larger margin (up to 20%). For hierarchical attributes, we show that our variable granularity approach based on Dewey encoded IDs performs much better (up to 300%) than the fixed granularity approach.

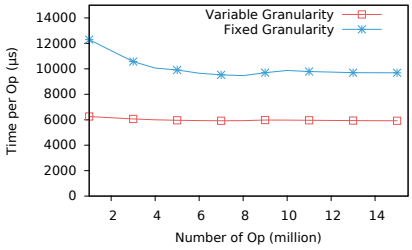
In the future, we plan to extend this work to a distributed environment. We also plan to allow more complex subscription queries that do not match based on a primary



(a) Overall performance



(b) SUBSCRIBE performance



(c) PUBLISH performance

Fig. 15: Performance of Pub/Sub in DualDB for high granular spatial data covering area of different radii

key conditions, where any of the existing secondary indexing techniques [29] can be used on top of our proposed frameworks.

Acknowledgements This project is partially supported by NSF grants IIS-1447826, IIS-1619463 and IIS-1746031.

References

1. Gemfire continuous querying. `https://pubs.vmware.com/vfabric5/index.jsp?topic=/com.vmware.vfabric.gemfire.6.6/developing/continuous_querying/how_continuous_querying_works.html`
2. Leveldb. http://leveldb.org/
3. Mongodb. https://www.mongodb.com
4. Project website for open source code. Http://dblab.cs.ucr.edu/projects/PubSub-Store/
5. Rocksdb. http://rocksdb.org/
6. Alsubaiee, S., Behm, A., Borkar, V., Heilbron, Z., Kim, Y.S., Carey, M.J., Dreseler, M., Li, C.: Storage management in asterixdb. Proceedings of the VLDB Endowment **7**(10) (2014)
7. Babu, S., Widom, J.: Continuous queries over data streams. ACM Sigmod Record **30**(3), 109–120 (2001)
8. Bhatt, N., Gawlick, D., Soylemez, E., Yaseem, R.: Content based publish-and-subscribe system integrated in a relational database system (2002). US Patent 6,405,191

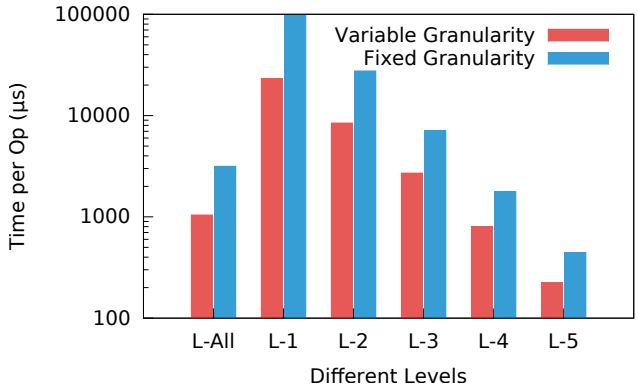


Fig. 16: Performance comparison in different levels of Hierarchy

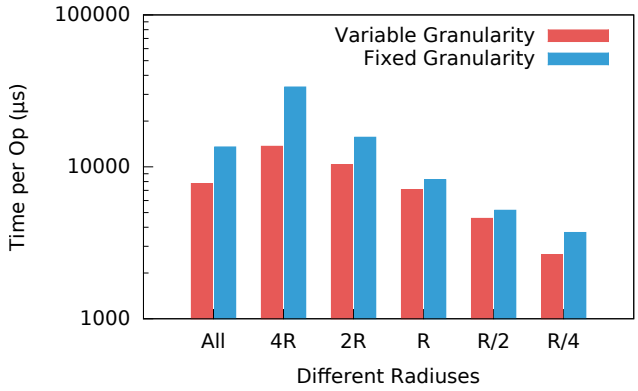


Fig. 17: Performance comparison for different radii

9. Carey, M.J., Jacobs, S., Tsotras, V.J.: Breaking bad: a data serving vision for big active data. In: Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, pp. 181–186. ACM (2016)
10. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Reiss, F., Shah, M.A.: Telegraphcq: continuous dataflow processing. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pp. 668–668. ACM (2003)
11. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. TOCS **26**(2), 4 (2008)
12. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: Niagaracq: A scalable continuous query system for internet databases. In: ACM SIGMOD Record, vol. 29, pp. 379–390. ACM (2000)
13. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing, pp. 143–154 (2010)
14. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys (CSUR) **35**(2), 114–131 (2003)
15. Feinberg, A.: Project voldemort: Reliable distributed storage. In: Proceedings of the 10th IEEE International Conference on Data Engineering (2011)

16. Fidler, E., Jacobsen, H.A., Li, G., Mankovski, S.: The padres distributed publish/subscribe system. In: FIW, pp. 12–30 (2005)
17. Garg, N.: Apache Kafka. Packt Publishing Ltd (2013)
18. George, L.: HBase: the definitive guide. O'Reilly Media, Inc. (2011)
19. Guo, L., Zhang, D., Li, G., Tan, K.L., Bao, Z.: Location-aware pub/sub system: When continuous moving queries meet dynamic event streams. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 843–857. ACM (2015)
20. Hanson, E.N., Carnes, C., Huang, L., Konyala, M., Noronha, L., Parthasarathy, S., Park, J., Vernon, A.: Scalable trigger processing. In: Data Engineering, 1999. Proceedings., 15th International Conference on, pp. 266–275. IEEE (1999)
21. Hendawi, A.M., Gupta, J., Shi, Y., Fattah, H., Ali, M.: The microsoft reactive framework meets the internet of moving things
22. Influxdb. https://www.influxdata.com/
23. Jacobs, S., Uddin, M.Y.S., Carey, M., Hristidis, V., Tsotras, V.J., Venkatasubramanian, N., Wu, Y., Safir, S., Kaul, P., Wang, X., Qader, M.A., Li, Y.: A bad demonstration: towards big active data. Proceedings of the VLDB Endowment **10**(12), 1941–1944 (2017)
24. Jacobsen, H.A., Muthusamy, V., Li, G.: The padres event processing network: Uniform querying of past and future eventsdas padres ereignisverarbeitungsnetzwerk: Einheitliche anfragen auf ereignisse der vergangenheit und zukunft. it-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik **51**(5), 250–260 (2009)
25. Kermarrec, A.M., Triantafillou, P.: Xl peer-to-peer pub/sub systems. ACM Computing Surveys (CSUR) **46**(2), 16 (2013)
26. Lakshman, A., Malik, P.: Cassandra: A decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
27. Madden, S., Shah, M., Hellerstein, J.M., Raman, V.: Continuously adaptive continuous queries over streams. In: Proceedings of the 2002 ACM SIGMOD international conference on Management of data, pp. 49–60. ACM (2002)
28. Oracle bitmap indexes. https://docs.oracle.com/cd/B10500_01/server.920/a96520/indexes.htm
29. Qader, M.A., Cheng, S., Hristidis, V.: A comparative study of secondary indexing techniques in lsm-based nosql databases. In: Proceedings of the 2018 International Conference on Management of Data, pp. 551–566. ACM (2018)
30. Qader, M.A., Hristidis, V.: Dualdb: An efficient lsm-based publish/subscribe storage system. In: Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM) (2017)
31. Schreier, U., Pirahesh, H., Agrawal, R., Mohan, C.: Alert: An architecture for transforming a passive dbms into an active dbms. In: Proceedings of the 17th International Conference on Very Large Data Bases, pp. 469–478. Morgan Kaufmann Publishers Inc. (1991)
32. Tatarinov, I., Viglas, S.D., Beyer, K., Shanmugasundaram, J., Shekita, E., Zhang, C.: Storing and querying ordered xml using a relational database system. In: Proceedings of the 2002 ACM SIGMOD international conference on Management of data, pp. 204–215. ACM (2002)
33. Tian, F., Reinwald, B., Pirahesh, H., Mayr, T., Myllymaki, J.: Implementing a scalable xml publish/subscribe system using relational database systems. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, pp. 479–490. ACM (2004)
34. Widom, J., Finkelstein, S.J.: Set-oriented production rules in relational database systems. In: ACM SIGMOD Record, vol. 19, pp. 259–270. ACM (1990)