DualDB: An Efficient LSM-based Publish/Subscribe Storage System

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

ABSTRACT

Publish/Subscribe systems allow subscribers to monitor for events of interest generated by publishers. Current publish/subscribe query systems are efficient when the subscriptions (queries) are relatively static – for instance, the set of followers in Twitter – or can fit in memory. However, an increasing number of applications in this era of Big Data and Internet of Things (IoT) are based on a highly dynamic query paradigm, where continuous queries are in the millions and are created and expire in a rate comparable, or even higher, to that of the data (event) entries. For instance moving objects like airplanes, cars or sensors may continuously generate measurement data like air pressure or traffic, which are consumed by other moving objects.

In this paper we propose and compare a novel publish/subscribe storage architecture, *DualDB*, based on the popular NoSQL Log-Structured Merge Tree (LSM) storage paradigm, to support high-throughput and dynamic publish/subscribe systems. Our method naturally supports queries on both past and future data, and generate instant notifications, which are desirable properties missing from many previous systems. We implemented and experimentally evaluated our methods on the popular LSM-based LevelDB system, using real datasets. Our results show that we can achieve significantly higher throughput compared to state-of-the-art baselines.

CCS CONCEPTS

•**Information systems** →**DBMS engine architectures;** *Key-value stores; Location based services;* Query optimization;

KEYWORDS

Log-Structured Merge Tree, LevelDB, Publish/Subscribe, NoSQL, Continuous Query, Instant Notification, Triggers, Big Data

ACM Reference format:
Mohiuddin Abdul Qader Vagelis Hristidis. 2017. DualDB: An Efficient LSM-based Publish/Subscribe Storage System. In *Proceedings of SSDBM '17, Chicago, IL, USA, June 27-29, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3085504.3085528

SSDBM '17, Chicago, IL, USA

978-1-4503-5282-6/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3085504.3085528

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, 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. As another application, an airplane continuously queries for data in its path such as turbulence, wind, air pressure, etc.

Challenges and requirements A key component of a Publish/Subscribe system is its storage module, which stores both the subscriptions and the publications. 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 these repetitive queries on the publications database. A problem 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. (3) 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.

Proposed work To support these properties we propose an efficient storage framework on top of LSM-based NoSQL databases. We argue that NoSQL databases – such as Cassandra [9], BigTable [3], AsterixDB [1] and LevelDB [7] – provide the right primitives on top of which we can design an efficient Publish/Subscribe storage system that addresses the above limitations. Specifically, LSM-based databases, compared to relational databases, offer fast write throughput and fast lookups on the primary key of a data entry.

To efficiently answer simple matching subscriptions, we propose a novel *DualDB* approach where both queries and data entries are stored in the same keyspace of the same database. We prefer to use *DualDB* instead of maintaining two separate databases, one for subscriptions (queries) and one for publications (data records) and thus suffer from a high rate of random disk accesses. We show that it is efficient compared to a baseline based on repetitive queries (*RepQueries*) mentioned above, when queries simply perform lookups, e.g., return all events (publications) inside my current cell id.

We compare our systems with a state-of-the-art pub/sub system Padres [6]. We chose Padres because it is open source and is easy to modify to fit our use-case. Padres supports query on historic/past data and uses the PostgreSQL database system. We show that Padres does not scale with the number of subscriptions and the system runs out of memory and crashes very quickly because they use an in-memory module to handle the subscriptions.

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

- We propose and implement a novel approach *DualDB* to efficiently support massive amount of highly dynamic subscriptions and publications on LSM-based storage systems (Section 4).
- 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 5). Our experiments show that our *DualDB* 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 6).

The rest of the paper is organized as follows. Section 2 reviews the related work. Then, Section 3 presents framework of the system, Finally, Section 7 concludes and presents future directions.

2 RELATED WORK AND BACKGROUND

Most of the academic work on pub/sub mainly has studied how to efficiently route the message through the distribAsterixDb recently added support for complex continuous queries using periodic repetitive executionfor pub/sub systems. There are variations of pub/sub systems supporting content or topic based subscription [5] Very few works studied the storage architecture of pub/sub systems. Padres is a popular open-source pub/sub system which supports subscriptions on future and historic data, using a PostgreSQL database inside each broker [8]. However, it does not scale with the number of subscriptions as we shown in our experiments.

Continuous queries in databases may be implemented using triggers [10]. However, they are only able to handle a very small number of triggers on a table [4], 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. Systems like NiagraCQ [4] proposed techniques to group continuous queries with similar structure, to share common computation. 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. AsterixDB recently added support for complex continuous queries using periodic repetitive execution [2].

Background on LSM tree and LevelDB. 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. All writes go to the in-memory component first and when filled, they flush into disk. A background process (compaction) periodically merges the smaller components to larger components as the data grows. A read (GET) on an LSM tree starts from Memtable and then goes to the disk compoenents until the desired entry is found, which makes sure the newest (valid) version of an entry will be returned. The SSTables of this LSM-based LevelDB are organized into a sequence of levels where Level-$(i+1)$ is 10 times larger than level-i in LevelDB. Each level (component) consists of a set of disk files (SSTables), each having a size around 2 MBs. The SSTables in the same level may not contain the same key (except level-0).

3 FRAMEWORK

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, $Subscription-JSON$)) 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 $T_{\min} = T_S - 10$ sec, $T_{\max} = T_S + 10$ 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, $Publication-JSON$)), i.e. an event occurred, we look for all the stored subscriptions to notify them if the ID (cell-id for moving

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

Operation	Description
SUBSCRIBE (*ID*, *Subscription−JSON*)	Write the Subscription in Subscription Storage with *ID* as primary key and return the list of matching and valid historic/past publications from Publication storage those had been published within T_{min}, T_{max} time interval.).
PUBLISH (*ID*, *Publication−JSON*)	Write Publication to storage with *ID* as primary key and return the list of matching and valid Subscribers who subscribed for this ID (i.e. topic/region) from Subscription storage.

objects or topic-id for content-based pub/sub systems) of the publication matches that of the subscription. 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.

We define ID as the Key that joins subscriptions and publications, and *Subscription−JSON* and *Publication−JSON* in JSON format as the value in our Key-Value Storage. They hold the attributes in Table 2. We will use there terms of Table 2 throughout the paper. 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.

Table 2: Set of Attributes and Terminologies

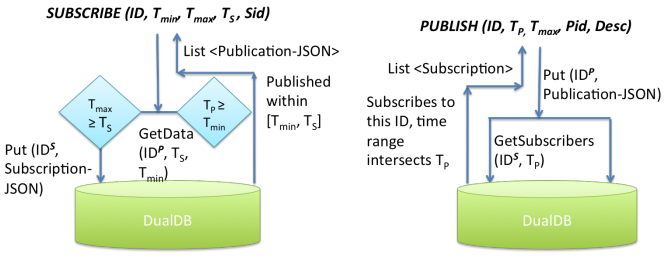
ID	cell-id/topic-id/product-id
T_S, T_P	Execution time of a Publication and Subscription
T_{min}, T_{max}	Time interval in Subscription. T_{max} is the expiry time for Subscription/Publication.
Sid, *Eid*	Subscription and Publication Identifier
Desc	Description text of a Publication
Subscription−JSON	Body of a Subscription in JSON { T_{min}, T_{max}, T_S, *Sid* }
Publication−JSON	Body of a Publication in JSON { T_P, T_{max}, *Pid*, *Desc* }

4 PROPOSED APPROACHES

We propose and implement *DualDB* approach to efficiently realize the API defined in Section 3, while providing instant response times to publications. *DualDB* approach maintains a optimized single database holding both subscriptions and data records/publications. To compare our approach with reasonable baseline, we also propose a baseline approach *RepQueries* which perform queries via repetitive channel which can not provide instant response. For both the approaches, we use the list storage mechanism described in Section 5, which maintains a lazily updated value list instead of a single value associated with a key (i.e. instead of single (key,value) pair we have fragmented value list for a key throughout the storage). This system efficiently remove the expired items from its storage during background compaction based on their expiry time (T_{max}).

4.1 Single Database Approach (DualDB)

As discussed in Section 2, previous works assume that the subscriptions are stored in memory, which is not realistic in our scenario. A key observation is that *if subscription queries and 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* Op. **(b) Processing of a *PUBLISH* Op.**

Figure 1: 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 1a and 1b show, respectively, how new query and new publication of events are processed in the proposed *DualDB*.

For every new subscription query, we must, in parallel, insert subscription in *DualDB* and query from *DualDB* for matching events/topics. The query $GetData\left(ID^P, T_S, T_{min}\right)$ 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. Since the postings list could be scattered in different levels (Section 5, *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.

For each new publication, we must, in parallel, check if an active subscription query matches it, and also insert it into the *DualDB*. The matching function $GetSubscribers\left(ID^S, T_P\right)$ 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 is similar to *GetData* operation moving down level by level in LSM storage to look for matching subscriptions.

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. Figure 2 show snapshots of the entries in the corresponding databases for the *DualDB* approach.

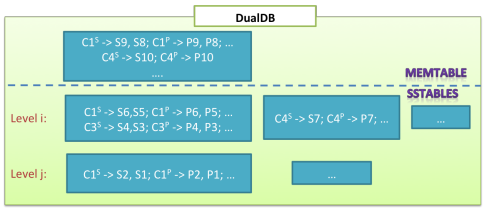


Figure 2: Snapshot of storage components for our LevelDB-style LSM database DualDB.

Here we can see that *DualDB* is holding posting lists of subscriptions (S1,S2,...) and publications (P2,P1,...) 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. This figure assume a leveled storage architecture (as in LevelDB and Cassandra); a stack-based architecture can be handled similarly.

4.2 Repetitive Queries Baseline (RepQueries)

DualDB approach discussed above support dynamic and massive amount of continuous queries on a storage framework holding lazily updated lists, which returns matching subscribers whenever an topic/event is published. This feature enables the system to support instance response to the user via continuous channel. To compare our system, we also prepared a baseline *RepQueries* which relies on repetitive channel and can not guarantee instance response. This approach is used in many pub/sub systems where they use a broker management as a middle-ware between 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 now responsible to detect duplicate data if some consecutive time range in the query intersects with each other resulting in return of duplicate publications. Also as the events are highly dynamic, some of them might expire in between the repetitive queries and user can potentially face missing data.

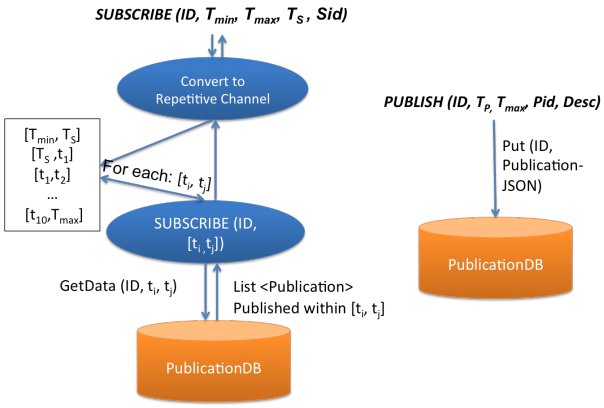


Figure 3: RepQueries Approach: Processing of *SUBSCRIBE* and *PUBLISH* Operation

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. To compare baseline with our proposed approaches, we converted each SUBSCRIBE operation into 10 repetitive query where we divide the time interval $[T_{min}, T_{max}]$ into 10 intervals. First it will issue a historic query on past data with time interval $[T_{min}, T_Q]$. Then it repetitively issue query and perform *GetData* operation after every $(T_{max} - T_Q)/10$ sec with time interval $[T_S, t_1]$, $[T_S, t_1]$, ..., and $[t_{10}, T_{max}]$. We consider this as equivalent to one SUBSCRIBE operation in instance-response approaches. The *GetData* operation follows the same procedure on Publication Database as in *DualDB*.

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 very slow for *RepQueries* approach than our other two approaches. Figures 3 shows, how a new subscription and a new publication of events are processed in the *RepQueries*.

5 ENABLE VALUE LISTS IN LEVELDB

The standard LevelDB implementation, as any key-value store, only support the storage of <key,value> pairs. However, to realize our pub/sub storage algorithms, we need to store <key, list<value» pairs, e.g., to store all publications for a single ID. 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 cell-id as key.

SSTables organized into levels in LSM tree based LevelDB and their compaction policy ensures only one unique key at 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 subscriptions with publications using this key. So, instead of one record associated with a key we will have a posting list. 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 is to perform 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. Now we have fragmented lists 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

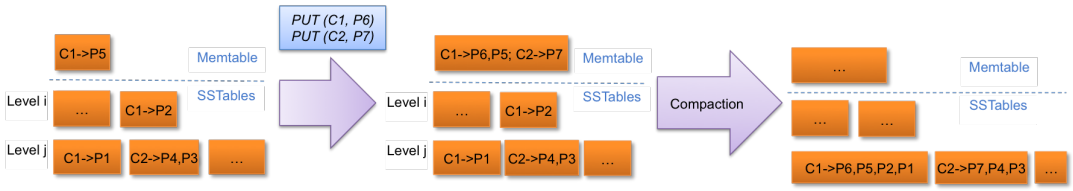


Figure 4: Writing data to Storage containing lazy updated value lists and their compaction.

fragmented lists. Lets illustrate the lazy update strategy on posting lists with an example.

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)*. Figure 4 depicts the state of the storage components (i.e. SSTables and Memtable) after these operation. We can see value list of C1 and C2 are fragmented in components of different levels. Then two operation PUT(C1,P6), PUT(C2,P7) are issued and we show in the Figure 4 how it affects the current lists and how these lists are lazily updated after 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 combined into larger lists.

6 EXPERIMENTS

6.1 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.

Data and Query Workload. We used Twitter streaming API to collect about 15 millions 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}) as 20s ahead of current time T_P.

To simulate a highly dynamic publish/subscribe model, we generated different workloads containing a mix of dynamically expiring subscriptions and publications 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 number of cells and about 15.3 million tweets, average number of tweets per cell is about 60 and as described, these tweets are converted into events and subscriptions.

Padres Baseline. In addition to the repetitive queries baseline (RepQueries) described in Section 4.2, we also compare to a popular Pub/Sub system, Padres. We understand that Padres 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. 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_Q, a 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).

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

$$S\{\left[class, eq, publication\right], \left[T, <, T_{max}\right], ..., \left[ID, eq, cid\right]\} \quad (2)$$

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).

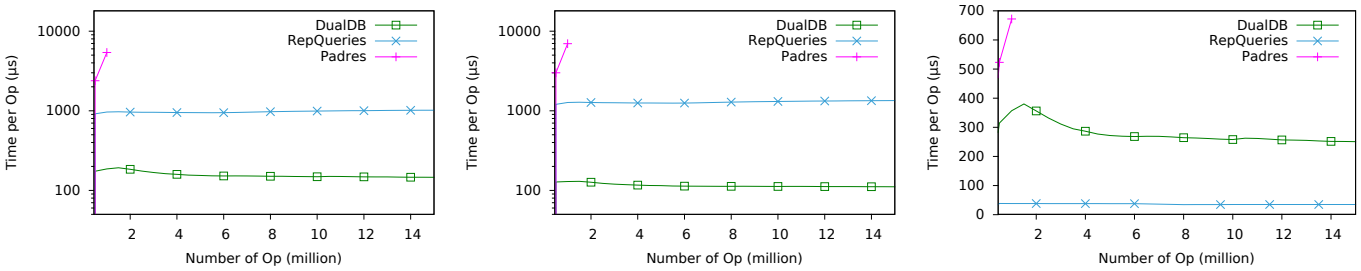
$$P\left[class, publication\right], \left[T, T_P\right], \left[desc, any\right], \left[ID, cid\right] \quad (3)$$

6.2 Experimental Results

We conduct our experiments on workloads for simple matching subscription with different subscription-to-publication ratios, which represent different use cases.

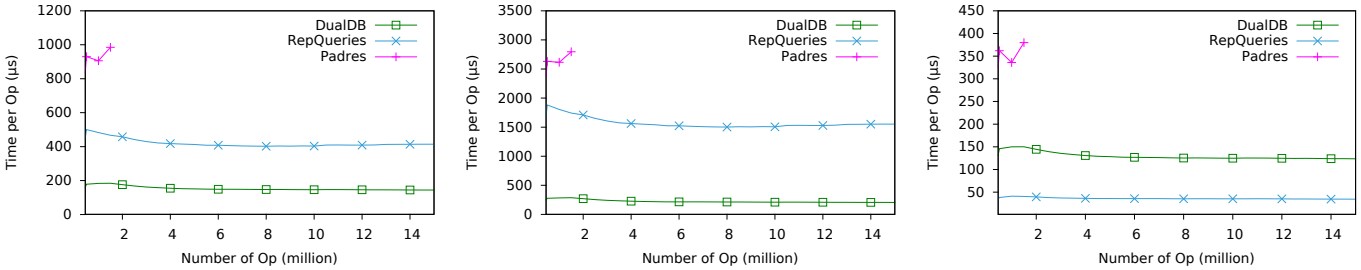
Simple Subscriptions. Figures 5 and 6 show the overall, SUBSCRIBE and PUBLISH performance of all systems Subscription heavy ($\frac{Subscription}{Publication} = 3$) and Publication heavy ($\frac{Subscription}{Publication} = \frac{1}{3}$) workloads, respectively. In all figures, we record the performance 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 5 and 6 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 variant. *DualDB* approach shows



(a) Overall performance (b) SUBSCRIBE performance (c) PUBLISH performance

Figure 5: Performance of different storage variants for simple subscription queries on Subscription Heavy Workload



(a) Overall performance (b) SUBSCRIBE performance (c) PUBLISH performance

Figure 6: Performance of different storage variants for simple subscription queries on Publication Heavy Workload

about 1000% better performance than *RepQueries* for Subscription heavy workload, and 300% better for Publication heavy workload.

We see in Figures 5 and 6 that Padres performs poorly not only compared to our DualDB approach, 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.

7 CONCLUSIONS AND FUTURE WORK

In this paper we present an efficient storage and indexing approach *DualDB* to achieve high throughput publish/subscribe on LSM-based databases where both the number of 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 implement our storage framework on top of the popular LSM-based LevelDB system and conduct extensive experiments using real datasets. The experimental results show that *DualDB* outperforms the state-of-the-art *Padres* pub/sub system (by up to 3000%) and also outperform the repetitive baseline *RepQueries* (by up to 1000%).

In the future, we plan to extend this work to a distributed environment and allow more complex subscription queries (e.g. subscriptions on hierarchical attributes, subscriptions that do not match based on a primary key conditions etc).

ACKNOWLEDGMENTS

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

REFERENCES

[1] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J Carey, Markus Dreseler, and Chen Li. 2014. Storage Management in AsterixDB. *Proceedings of the VLDB Endowment* 7, 10 (2014).
[2] Michael J Carey, Steven Jacobs, and Vassilis J Tsotras. 2016. Breaking BAD: a data serving vision for big active data. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM, 181–186.
[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *TOCS* 26, 2 (2008), 4.
[4] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD Record*, Vol. 29. ACM, 379–390.
[5] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)* 35, 2 (2003), 114–131.
[6] Eli Fidler, Hans-Arno Jacobsen, Guoli Li, and Serge Mankovski. 2005. The PADRES Distributed Publish/Subscribe System.. In *FIW*. 12–30.
[7] Google Inc. 2017. LevelDB. http://leveldb.org/. (Feb 2017).
[8] Hans-Arno Jacobsen, Vinod Muthusamy, and Guoli Li. 2009. 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 (2009), 250–260.
[9] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (apr 2010), 35–40.
[10] Jennifer Widom and Sheldon J Finkelstein. 1990. Set-oriented production rules in relational database systems. In *ACM SIGMOD Record*, Vol. 19. ACM, 259–270.