Bloom Filters in Incremental Execution

William W. Ma
University of Chicago
williamma@uchicago.edu

ABSTRACT

Due to an ever increasing amount of data, it has never been more important to efficiently process the data. To maximize efficiency, batched lazy execution is the most efficient. That efficiency, however, comes at the cost of high latency. To minimize latency, eager execution minimizes latency as much as possible at the cost of efficiency. To strike a balance between efficiency and latency, incremental batch execution allows the user to decide how much latency they are willing to tolerate and then maximizes efficiency within those constraints. To do incremental batch execution properly, state has to be persisted. While it would be obvious that putting bloom filters on the state would improve efficiency, we show that bloom filters can lead to a slight performance degradation.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
William W. Ma. 2018. Bloom Filters in Incremental Execution. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

As the amount of data quadruples [3], it is increasingly important to improve the efficiency of data processing in order to reduce the carbon footprint of computing while still maintaining and improving the daily lives of people through computing. We can simultaneously reduce latency and maintain efficiency through the efficient processing of streaming data. Traditional approaches to streaming data have focused on either lazy or eager execution. Lazy execution maximizes efficiency but has a high latency cost, which is not acceptable in some use-cases. Eager execution, on the other hand, minimizes latency but is not efficient.

Recently [4], incremental batch execution has been proposed. This allows users the opportunity to choose a middle ground between lazy and eager execution. Thus, users are given the ability to choose their latency or efficiency requirements. However, to have incremental batch execution, the query state will need to be persisted onto stable storage. With this requirement, the implementation of the physical operators that interact with the state can dramatically affect the performance of the query execution and, in turn, efficiency of the data processing.

Listing 1: Example count distinct from TPC DS

```
SELECT  *
FROM (SELECT AVG(price) B1_LP
            ,COUNT(price) B1_CNT
            ,COUNT(distinct price) B1_CNTD
      FROM sales
      WHERE quantity BETWEEN 0 AND 5
        AND (price BETWEEN 11 AND 21
             OR coupon BETWEEN 460 AND 1460
             Or cost BETWEEN 14 AND 34)),
     ...
LIMIT 100;
```

In this paper, we simulate the execution of a `DISTINCT COUNT` query and explore different implementations of a `DISTINCT COUNT`. We chose to model a `DISTINCT COUNT` query for two reasons. Firstly, they are a common type of query, such as query 28 partially reproduced in listing 1. Furthermore, it is a simple query that has the ability to capture a wide range of behaviors, which can affect data processing efficiency and latency. These behaviors include data distribution, data size, and intermediate query state size. We discuss the choice of a `DISTINCT COUNT` query in section 2.1. In terms of implementations, we explore the performance benefits of bloom filters [1]. The intuition behind the addition of a bloom filter is that they would reduce latency by reducing IO time to reach disk.

To profile the performance of different `DISTINCT COUNT` implementations, we created RoomDict (Rapid Out-Of-Memory Dict), a python library that provides a hash map with support for arbitrary storage, caching policies, and membership tests, such as bloom filters. The code is provided on github [2].

Thus, our contributions for this paper are as follows:

- Exploring the performance trade offs of different physical implementations for a `DISTINCT COUNT` query within an incremental batch execution context.
- Provide a simple python library with a standard key-value interface that implements a hash map, which is able to support multiple arbitrary storage devices, cache policies, and membership tests.

We structure the rest of the paper as follows. First, Section 2 discusses how we can reduce select-project-join-aggregate queries to maintaining hash maps. Then, Section 3 goes over the architecture of RoomDict and how it facilitates different physical implementations. Next, we show the experiments used to demonstrate the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Woodstock '18, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

performance of different physical implementations in section 4. Finally, we discuss future work in section 5.

2 LOGICAL MODEL

In this section, we introduce how `DISTINCT COUNT` queries are able to capture the entire range of behaviors that are present in incremental batch execution of select-project-join-aggregate queries. Additionally, we also discuss how only maintaining the intermediate state of a `DISTINCT COUNT` query can still capture the performance envelope of executing the entire `DISTINCT COUNT` query. The combination of these two ideas allows us to use the maintenance of the intermediate state of a `DISTINCT COUNT` query, a hash map, to capture the performance envelope of all select-project-join-aggregate queries.

2.1 Reducing General Queries to DISTINCT COUNT Queries

In this paper, we focus on improving the efficiency of select-project-join-aggregate queries. In doing so, we can safely ignore simple select and filter type queries because improving the efficiency of those is well studied within the context of query compilation and vectorized execution of database queries. For simplicity, we focus our attention to only include equality joins and exclude other types of joins.

Thus, our focus on (equality) joins and aggregations implies that the state that our queries have to maintain during execution is a hash map. This hash map would map the objects that we have seen so far to some sort of metadata. For (equality) joins, the mapping would be from the join keys we have seen so far to the records that have the given join keys. For aggregations, the mapping would be from the aggregation key to the intermediate value of the aggregation given the records that the execution has gone through up until this point in the execution. For `DISTINCT COUNT` queries specifically, the mapping would be to the distinct keys it has seen so far to the count of those keys it has seen thus far in the execution. Thus, it is obvious that we can model select-project-(equality) join-aggregate queries as `DISTINCT COUNT` for incremental batch execution, since the main difference from incremental batch execution to traditional batch execution is the fact that we have to maintain the intermediate state between successive, incremental executions. This intermediate state, a hash map, is the same across all of these queries.

Furthermore, the performance characteristics of select-project-(equality) join-aggregate queries can be similarly captured within `DISTINCT COUNT` queries. Specifically, we expect the performance to be tied to both the amount of data and the distribution of the data within every incremental batch execution of the queries.

First, it is trivial that the performance is tied to the amount of data being executed in every incremental batch execution. Additionally, the amount of data in the intermediate state will grow linearly with respect to the number of unique keys seen so far for aggregations, including `DISTINCT COUNT`. However, for joins, the amount of data in the intermediate state will grow linearly with respect to the number of rows seen so far. To simplify our analysis, we assume that the number of join keys is approximately the same as the number of rows for joins. This assumption is a fair assumption as all of the joins within TPC-H and TPC-DS are joined on id type fields. Thus, the number of rows seen in a join is approximately equal to the number of unique keys seen so far. As a result, we can assume that there is roughly a constant amount of work to be done for every new value with respect to the amount of time to process an increasing amount of data. Thus, the performance of both `DISTINCT COUNT` and general select-project-(equality) join-aggregate queries is the same in the face of increasing the amount of data being processed.

Secondly, the performance is also tied to the distribution of the data. This is due to the fact that for both (equality) joins and aggregations, the intermediate state is a hash table. This hash table may grow beyond the available amount of memory for the query's set memory allocation. Thus, the hash table will be forced to spill to disk. As a result, the access time for all keys may not be identical. Therefore, as the hash map grows beyond the size of memory, the access time of the hash map is not constant, which leads to a difference in execution time when we compare this data set to a similar data set, but a smaller number of unique keys. As a result, the number of unique keys, or the distribution of the data, can affect performance.

Thus, all of the select-project-(equality) join-aggregate queries can be captured by `DISTINCT COUNT` queries, if we make the assumption that the number of join keys is approximately equal to the number of rows in the table.

2.2 Reducing DISTINCT COUNT Queries to Hash Map Maintenance

While there is a lot more to incremental batch execution of a `DISTINCT COUNT` query, we show that the main factors that affect performance, amount of data and the distribution of the data, can be captured by maintaining only the hash map that makes up the intermediate state. Specifically, we try to the performance characteristics of the overall latency for the entire incremental batch execution. For our purposes, we assume that the hash map attempts to stores everything in-memory, but is allowed to spill parts of the hash map to disk if there is not enough memory to store the hash map.

Firstly, it is obvious that as the amount of data increases, assuming a constant data distribution, the execution time will increase for both incremental batch execution of a `DISTINCT COUNT` query and simply maintaining the hash map of the `DISTINCT COUNT` query. While there are other considerations as the data increases in a real database, such as garbage collection, data compaction, and transaction management, we treat these as constants with respect to the increase in the amount of data being processed. Thus, we can expect to see the same performance patterns in both increasing the amount of data being processed in a `DISTINCT COUNT` and only maintaining the hash map of the intermediate state of a `DISTINCT COUNT` query.

Secondly, while we keep the data size the same, data distribution changes should have the same effects on both executing the entire `DISTINCT COUNT` query and only maintaining the intermediate state. This is fact relies on the assumption that we always have as much of the hash map in memory as we are allocated. Thus, there is no difference in how we access the hash map in both cases.

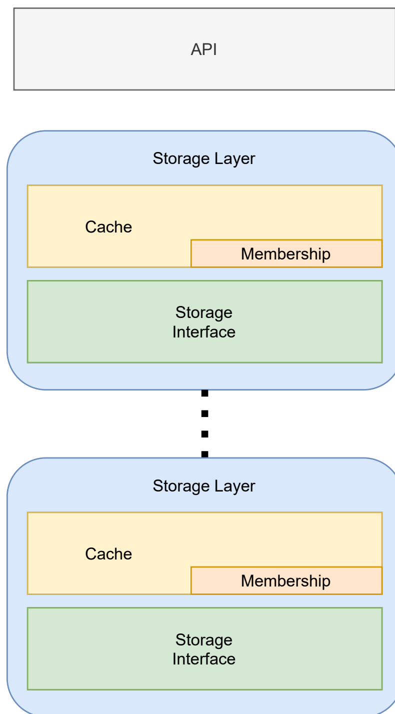

Figure 1: Logical architecture diagram of a RoomDict instance

However, there may be instances for when the entire hash map is persisted in memory and read back in at a later time. While we believe that this is a valid use-case, it makes more sense to have at least parts of the hash map in memory if memory is partitioned between all queries. However, this ultimately comes down to the architecture and design of each specific database. Furthermore, as long as both executions make the same assumption, we expect no significant performance deviations. Thus, data distributions effect on performance should be the same for both executing the entire `DISTINCT COUNT` query and only maintaining the intermediate state of the query, as long as the same assumptions are made for both.

3 PHYSICAL MODEL

In this section, we discuss the architecture of a RoomDict instance. RoomDict, on its own, only provides the building blocks and high level links. With this, users are able to customize their hash map to spill to arbitrary storage devices, use different caching policies, and different membership tests. Furthermore, the user is able to mix and match these lower level policies to build individualized "storage layers." These storage layers are the building blocks that the user creates and specifies the order of to customize their hash map. RoomDict takes the user's specification and "builds" the storage layers and connects them together.

3.1 RoomDict Instance Architecture

Every RoomDict instance implements the standard python dictionary interface, which is the `MutableMapping` abstract class in python. To instantiate a RoomDict instance, the user has to specify each "storage layer," which consists of a cache policy, membership test, and storage device. The defaults are "infinite" cache for the caching policy, no membership test, which implies that the storage layer will have to go to storage to check for membership, and disk storage.

The order of the storage layers implies the order in which the hash map will spill over. The first layer stores only as many keys as specified by the caching policy. When a user wants to add or update, which are implemented as a delete and insert, a key to the RoomDict hash map, the API layer will pass the new key to the first storage layer. If the first layer is full, RoomDict will use the caching policy to evict the key and pass the evicted key to the second layer. This continues until either the key is inserted into a storage layer or all storage layers are full, with respect to their caching policies, and the discarded key from the last layer is discarded.

When user wants to retrieve a key in the RoomDict hash map, the API layer passes the key to the first storage layer. This storage layer checks if it passes the membership test. Since we are currently only using bloom filters as our membership test, we are guaranteed no false positives but may have false negatives. Thus, if the membership test returns that it is in the storage device, the key is retrieved. Otherwise, we still have to go to the storage device to check if the key is in the current layer. If the key is confirmed to not be in the first storage layer, the key is passed to the next storage layer to check if it is there. This repeats until we have either found the key or checked all storage layers.

3.2 RoomDict Implementation

Since the storage layers are all relatively independent, the API layer coordinates all of the different storage layers. This coordination has minimal overhead because it does not have to pass any key-value pairs between the storage layer and the API. Only the keys are passed around, except when returning a key-value pair to the user.

For the purposes of this paper, we have implemented the following. For caching policies, we implemented an LRU cache and an "infinite" cache, that is a "cache" with no size limit.

For membership tests, we use pybloom3, a bloom filter in python, and a dummy membership test that always returns false. This default is valid because bloom filters only has false negatives. Thus, this dummy membership test is essentially a bloom filter with a 100% false negative rate.

For storage interfaces, we use standard python dictionaries for in-memory, `shelve` from the standard library for disk storage, and a storage layer that can simulate arbitrary IO latency. To work around the fact that `shelve` stores the keys in-memory, which would not properly represent our workload, we reload the `shelve` file from disk every time we need to access it. Our storage layer that simulates arbitrary IO latencies is simply an in-memory dictionary that sleeps for a user defined amount of time before any function executes. This simple model allows us to get a rough approximation of different storage device IO latencies. While we could add noise or have different latencies for different functions (e.g., longer latency

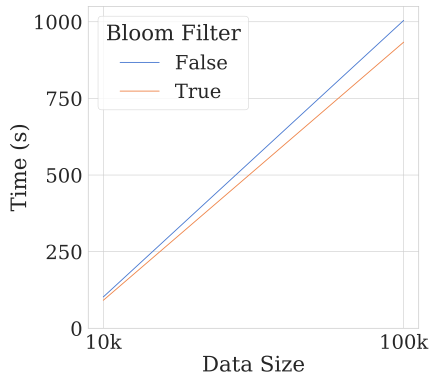

Figure 2: Time w/diff. amounts of data

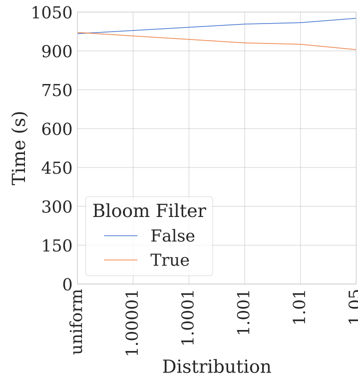

Figure 3: Time w/diff. latencies

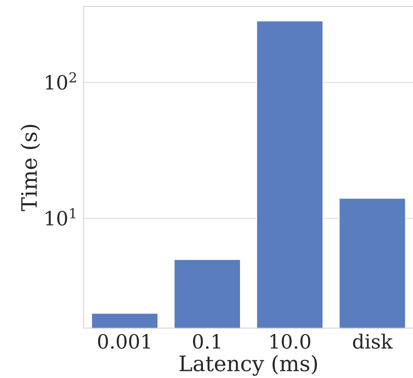

Figure 4: Time w/disk & diff. latencies

for checking membership as compared to key-value retrieval), for the sake of simplicity, we use a constant amount of latency for all functions and for all executions.

4 EXPERIMENTS

For our experiments, we used RoomDict with an in-memory LRU cache and either on disk or arbitrary storage with an "infinite" cache. This configuration ensures that we always have some keys in-memory and there may be other keys in storage. We also are ensured that no keys will be discarded as the disk or arbitrary storage captures all the remaining keys.

With these RoomDict instances, we look at the difference in run time between using and not using a bloom filter when we vary the data distribution, storage device latency, amount of data processed, cache size, and false negative rate of the bloom filter. To do these benchmarks, we used an Intel NUC with more than 8 GB of memory. Our experiments never exceeded 4 GB of memory so we do not have to deal with issues of virtual memory. Additionally, RoomDict is only single threaded and Intel NUCs have at least 2 CPUs so there should not be any issues with scaling.

In all of our figures, we ensure that all values of the dimension we are varying have the same distribution in the dimension we are not varying. Then we simply take the average over the dimension we are varying. With this, we ensure that the performance patterns that we see are not due to some other variable in the specific RoomDict configuration that we chose.

We simulate a `DISTINCT COUNT` intermediate state workload by generating a random set of keys following either a uniform or zipf distribution. We then use a RoomDict instance to count the number of occurrences for each key.

At a high level, our experiments show that bloom filters cause too much overhead relative to modern IO devices, which makes using bloom filters result in less performance.

4.1 Varying Data Parameters

In terms of data parameters that we have at our disposal to vary, we can vary the amount of data that is processed or the distribution of the data that we are processing.

Distribution	No Bloom Filter	Bloom Filter
Uniform	966 s	971 s
Zipf scale 1.0001	1003 s	931 s

Table 1: Run times (s) of different distributions

First, from figure 2, we can see that as the amount of data being processed increases, the amount of time to process the data also increases. The main difference we want to note is the fact that using a bloom filter leads to a shorter run time regardless of data size. However, the difference between using a bloom filter and not using a bloom filter goes down, relatively. Specifically, not using a bloom filter with 10,000 keys leads to a performance degradation of 15% but, 100,000 keys, the performance degradation is only 7%. This difference is reasonable because we are using an automatically scaling bloom filter, where the bloom filter increases in size to maintain an error rate as the amount of data the bloom filter has seen increases. Thus, with a larger data set, there will be more overhead with the bloom filter, as it has to increase in size more often. While bloom filters do lead to performance improvements when averaged over different data sizes, this pattern does not continue when averaging over different variables.

One such parameter that we are averaging over is the data distribution. However, when we look at the difference between using and not using a bloom filter across different data distributions in figure 3, where the distributions are either uniform or using a zipf with zipf scale factor shown on the x-axis. we see that the difference is minimized in a uniform distribution. We note the differences in table 1. The reason that the uniform data distribution results in a longer running time by using a bloom filter as opposed to not using a bloom filter is due to the fact that the uniform distribution is a continuous distribution. Thus, the keys generated are nearly all distinct from each other. However, the zipf distributions are discrete distributions. Thus, the scale factor determines how likely the keys are going to overlap with each other. Given this difference, if all or a majority of the keys are distinct, then the overhead to calculate the hashes and maintaining the bloom filter is not worth it. While having all distinct keys may not happen when doing aggregations, having all distinct keys is a certainty in TPC-H joins as TPC-H joins all occur over unique id keys. Furthermore, we see a larger reduction in running time when there are less distinct keys (i.e., higher zipf scale factor) when using a bloom filter as opposed to not

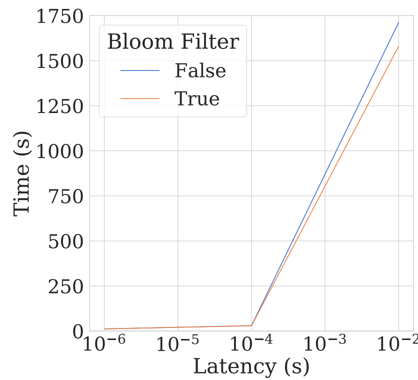

Figure 5: Time w/diff. device latencies

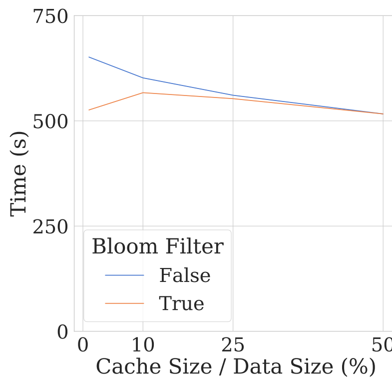

Figure 6: Time w/diff. cache ratios

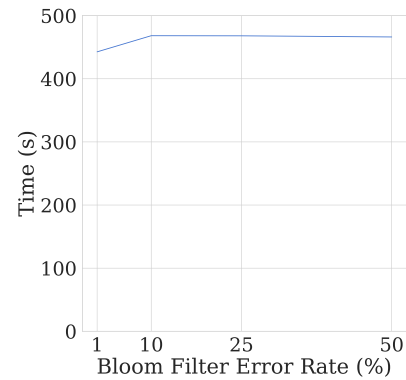

Figure 7: Time w/diff. error rates

using a bloom filter. This is due to the fact that the bloom filter can further reduce IO operations that need to be done. This reduction is on top of the reduction that comes as a benefit of having a cache. Additionally, with less distinct keys, there is less overhead needed to maintain the scalable bloom filter because there will be less times that we have to scale the bloom filter.

Overall, we find that the usage of a bloom filter leads to lower performance due to the fact that bloom filters have higher overhead than IO latencies.

4.2 Varying Device Parameters

When changing the device parameters, we have the ability to change the storage latency and cache size. Before diving into the effects of different storage latencies, we first look at the approximate storage latency of using disk storage. We can see from figure 4 that the runtime for disk storage is between that of a 0.1 ms and 10 ms simulated latency. However, this does not prove the assumption made in section 3.2 because in both the disk storage and the arbitrary storage engines, we simulate the IO time to be constant for every operation. Disk operations have to read the entire file in from disk and the arbitrary storage sleeps for an arbitrary amount of time. While the file does grow over time, most distributions have many "hot" keys, which means that the file on disk does not grow after a certain burn in period. Although this could be the case, more experiments are needed to prove that this is the case. While it is not precisely clear that our simulated times reflect actual storage media times 100%, this does show that the simulated times are within the ballpark and are good enough to use for high level analysis.

When we do vary the simulated storage latency, we notice that the runtime increases with the storage latency in figure 5. We can see that there is an linear increase in runtime as the simulated storage media latency increases. Unsurprisingly, the difference between using a bloom filter and not using a bloom filter also increases exponentially. This is expected because the bloom filter allows us to reduce one IO when looking for a key. However, this difference may be artificially inflated due to the fact that we are using the same IO latency for checking if a key exists. While it may be true that the difference is artificially inflated, the difference will still exists as long as we have to go to the (simulated) storage media to check for the existence of a key. It is worth noting that only at a simulated storage media latency of $10^{-6}s = 1\mu s$ is the overhead of using a bloom filter more expensive than not using a bloom filter. However, the difference in runtime is $0.1s$. Since SSD and hard drives have a storage latency of well beyond $1\mu s$, the difference can be safely ignored for practical applications.

In addition to changing the storage latency, we are also able to change the cache size. From figure 6, we can see that as the relative cache size increases, the run time decreases. Furthermore, as the cache size increases, the difference between using a bloom filter and not using a bloom filter also decreases. This is not surprising because as the cache size increases, the bloom filter code path is hit less. Thus the difference between using a bloom filter and not using a bloom filter will also decrease proportionally. It is worth noting that because the difference between using a bloom filter and not using a bloom filter when the cache size is 50% of the data size is 1 second implies that the majority of the data can fit within the cache, for most distributions tested.

4.3 Vary Implementation Details

Finally, we adjust the bloom filter error rate that we enforce. With the results in figure 7, we can see that there is a very small difference between an error rate of 10%, 25%, and 50%. Additionally, even the difference between 1% and 10% is only a 5% performance degradation. This is important because then we can simply set a larger error rate on the bloom filter without much worry for performance. However, this is assuming that we do not attempt to go for the marginal gains and reduce the error rate down below 1%, as it appears that the bloom filter error rate performance degrades exponentially as the error rate approaches zero.

5 FUTURE WORK

For future work, there are many paths that we can explore down the line. In terms of query latency predictions, we can use bloom filters as a way to smooth the difference between having a hash table entirely in memory and a hash table split between in memory and on different storage media. Borrowing from the patterns that we saw in figure 2 and figure 5, it appears that the patterns that we will see should be relatively easy to approximate, as they will

be either linear or exponential models. Both of which have been extensively studied in machine learning.

6 CONCLUSION

From this work, we have seen that using bloom filters are not always the best choice due to the overhead of maintaining the bloom filter and hashing keys. However, this overhead is only too expensive when the bloom filter is unnecessary. When a bloom filter is actually used (i.e., when the elements are not all distinct) a bloom filter can help reduce the run time of `DISTINCT COUNT` queries and, in turn, all incremental batch execution queries.

REFERENCES

[1] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.
[2] William W. Ma. 2021. RoomDict. https://github.com/williamma12/RoomDict.
[3] D Reinsel, J Gantz, and J Rydning. 2018. *Data age 2025. The digitization of the world: From edge to core. An IDC White Paper# US44413318.*
[4] Dixin Tang, Zechao Shang, Aaron J Elmore, Sanjay Krishnan, and Michael J Franklin. 2019. Intermittent query processing. *Proceedings of VLDB Endowment* 12, 11 (2019).