Cquirrel: Continuous Query Processing over Acyclic Relational Schemas

Qichen Wang[†], Chaoqi Zhang[†], Danish Alsayed[†], Ke Yi[†]
Bin Wu[‡], Feifei Li[‡], Chaoqun Zhan[‡]
Hong Kong University of Science and Technology[†] Alibaba Group[‡]
qwangbp@cse.ust.hk,{czhangci,dimialsayed}@connect.ust.hk,yike@cse.ust.hk
{binwu.wb,lifeifei,lizhe.zcq}@alibaba-inc.com

ABSTRACT

We will demonstrate Cquirrel, a continuous query processing engine built on top of Flink. Cquirrel assumes a relational schema where the foreign-key constraints form a directed acyclic graph, and supports any selection-projection-join-aggregation query where all join conditions are between a primary key and a foreign key. It allows arbitrary updates to any of the relations, and outputs the deltas in the query answers in real-time. It provides much better support for multi-way joins than the native join operator in Flink. Meanwhile, it offers better performance, scalability, and fault tolerance than other continuous query processing engines.

PVLDB Reference Format:
Qichen Wang, Chaoqi Zhang, Danish Alsayed, Ke Yi, Bin Wu, Feifei Li and Chaoqun Zhan. Cquirrel: Continuous Query Processing over Acyclic Relational Schemas. PVLDB, 14(12): 2667 - 2670, 2021.
doi:10.14778/3476311.3476315

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/hkustDB/Cquirrel-release.

1 INTRODUCTION

Query evaluation on a static database is a well studied problem. In many emerging applications, queries are evaluated on a database that is being continuously updated. Examples include online data analytics, stock price prediction, sensor monitoring, network traffic analytic, etc. During Singles' Day Global Shopping Festival of Alibaba [12], huge volumes of sales data are being collected, and it is very important to monitor various statistics (often in the form of SPJA queries) in real time, in order to make informed decisions. In these applications, updates to the database are being made at high speeds and the query processing system must maintain the query answer with high throughput and low latency. Another important application of continuous query processing is materialized view maintenance [5, 8]. A materialized view is nothing but the results of a predefined query, which can significantly reduce the cost of query evaluation, when the view is part of a bigger query. Materialized views are supported by most modern database systems. There is an extensive body of work on query answering using views [3, 11], while it is obvious that the view has to be up-to-date.

Problem definition. Let db be the contents of the current database and $Q(db)$ denote the results of evaluating query Q on db. For an update u to db, where u can be either the insertion or deletion of a tuple, we write $db + u$ as the new database instance after applying u. To facilitate the computation, some data structure on db, denoted $\mathcal{D}(db)$, can be maintained. Depending on the application, there are two output modes. (1) Delta enumeration: Given $\mathcal{D}(db)$ and u, the system should output ΔQ, i.e., all differences between $Q(db)$ and $Q(db + u)$. (2) Full enumeration: Given $\mathcal{D}(db)$, all query results in $Q(db)$ can be enumerated with constant delay [4].

Our demonstration system. We present Cquirrel (Continuous query processing over acyclic relational schemas), a system for continuous query processing built on top of Flink [6]. Cquirrel specifically targets at queries with multi-way foreign-key joins over an acyclic schema. The default output model of Cquirrel is delta enumeration. The user first registers a query on an initially empty database. An update sequence (containing insertions and deletions) are then fed into Cquirrel in the form of a DataStream of Flink. As updates are being processed, deltas of the query answer are generated as an output DataStream in real time. In our demo, we feed the output DataStream to a web-based user interface for rendering; in other applications, the output DataStream can be consumed by another Flink application, or any other consumer.

A schema is (foreign-key) acyclic if the graph formed by the primary-foreign key relationships is a directed acyclic graph (DAG). Acyclic schemas are very common in schema design [1]. For example, the TPC-H scheme is such one. In fact, it is considered problematic if the primary-foreign key relationships contain a cycle [13], which would inevitably result in the foreign-key constraint being violated when the underlying database is updated. Cquirrel supports any selection-projection-join-aggregation (SPJA) query over an acyclic schema, provided that all join conditions are between a primary key and one of the foreign keys referencing the primary key. Note that all join conditions in the TPC-H queries satisfy this requirement. The aggregation may optionally have a group-by and an order-by. The updates can be made to any relation in an arbitrary fashion, although the cost for handling the updates will depend on λ, the *enclosureness* of the update sequence [14]. Although λ can be high in the worst case, for most real-world update sequences, λ is a constant. In particular, if the updates are first-in-first-out (i.e., tuples are deleted in the order they are inserted), which includes the sliding-window model as a special case, then $\lambda = 1$.

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476315

Cquirrel uses the newly developed algorithms from [14] to achieve the $O(\lambda)$ update time. In addition, Cquirrel has the following distinctive features compared with other continuous processing systems.

Holistic processing of multi-way joins. Cquirrel considers a multi-way join as one operator. The system maintains index structures based on the entire join structure. Given a query, Cquirrel will first construct the foreign-key graph of the given query, which must be a DAG. Then Cquirrel handles updates level by level following the DAG. Compared with the traditional query processing technique that decomposes a multi-way join into multiple two-way joins, our approach does not need to materialize the intermediate join results, which significantly improves both time and memory consumption. In particular, the native join operator provided by Flink only supports two-way joins, so the user has to manually decompose a multi-way join into many two-way joins, and maintain the intermediate join results as DataStreams. As a result, Flink only supports continuous query processing in the sliding-window model over two-way joins; queries involving multi-joins are only supported in the tumbling-window model, or a sliding-window with a large sliding step size, which is not truly “continuous”. In contrast, Cquirrel processes the updates as they arrive in real time; for the special case of the sliding-window model, this means that deltas are generated as the window slides continuously.

Parallelism. In building Cquirrel, we have parallelized the algorithm in [14] to achieve a high throughput. Parallelization is done both vertically and horizontally. The index structure for each relation are hash-partitioned by the join key, which can then be handled by different workers in parallel. Tuples in different relations do not need to be co-partitioned in our design , so Cquirrel does not need to replicate any tuples nor index structures. The overall load will remain consistent, regardless of the number of workers. Furthermore, the index structures for different relations are also handled by different workers following the structure of the join DAG with no backward communication. This allows us to carry out the maintenance processes on different relations in parallelized pipelines. Thus, Cquirrel can easily scale to very high volumes of updates. Figure 1 shows the performance of Cquirrel with different levels of parallelism. Note that, however, due to different processing speeds at different workers, the output DataStream (the deltas) may not follow exactly the same order as the input DataStream (the updates). Specifically, if there are two updates u_1, u_2 that have arrived in this order, it is possible that we see the delta caused by u_2 before the delta of u_1. However, even when this happens, Cquirrel ensures eventual consistency, i.e., the sum of all deltas must be correct.

Integration with Flink. When the user registers a query, Cquirrel compiles the query plan into Flink code, which is then run by the Flink engine. In some sense, Cquirrel can be considered as “just” a query compiler. However, this modular design allows us to inherit all the benefits of Flink, which includes ultra-low latency, fault tolerance, and elasticity. In particular, the Flink engine ensures exactly-once semantics, so the query processing results are guaranteed to be correct eventually, although we may observe occasional inconsistencies while the update stream is being processed, as noted earlier. Finally, it is also easy to integrate Cquirrel into the broader Flink ecosystem, e.g., to train a machine learning model over changing data.

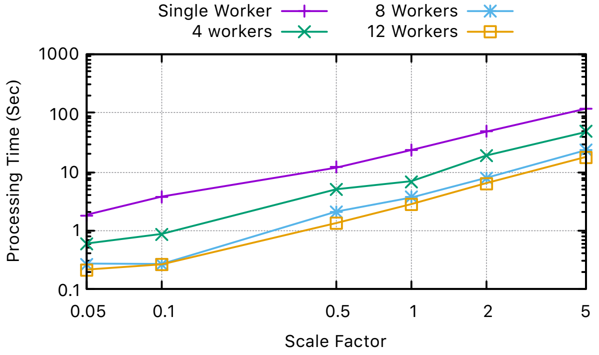


Figure 1: Running time on TPC-H Query 5 with different Parallelism and Scale Factor

Comparison with other systems. The above features give Cquirrel a distinctive position compared with other continuous query processing systems. Existing systems like DBToaster [2], Dynamic Yannakakis[10], and Trill [7] are centralized, thus do not scale well with high volumes of updates. In fact, Cquirrel can offer 2x to 80x improvements even when running with one worker, due to the optimized algorithm and index structure design. None of the existing systems offers any fault tolerance, either. However, some of them support a broader class of queries, where the join conditions may not be between a primary key and a foreign key.

2 CQUIRREL BY EXAMPLE

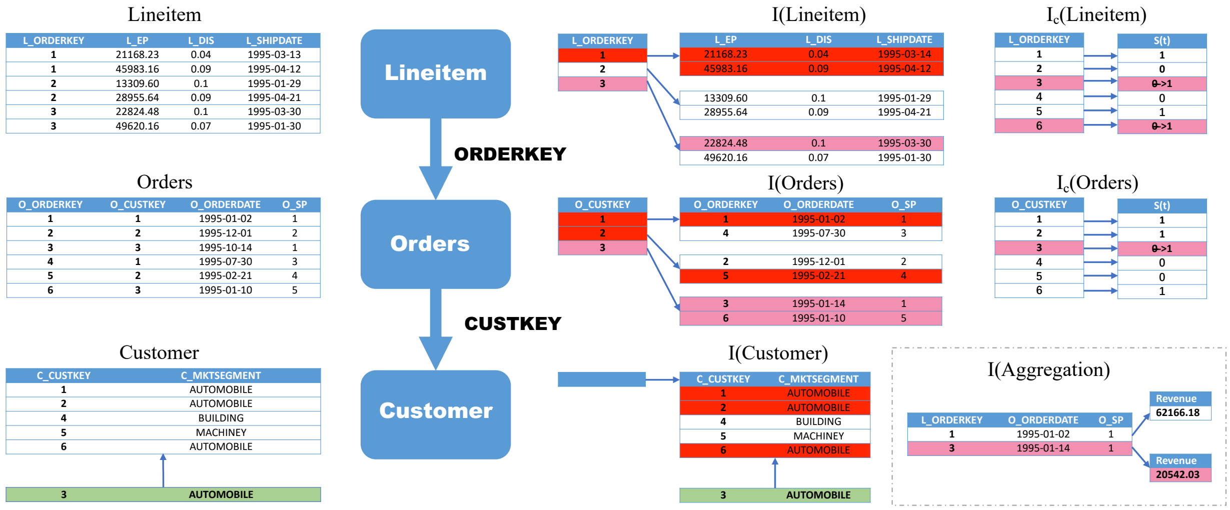
Our demo system will allow the user to provide SQL queries (for ease of demonstration, the schema will be fixed to be the TPC-H schema) through a web-based user interface, which also visualizes the query results in real time as updates are being processed.

Below, we use the following query (TPC-H query 3) to walk through the query processing stages in Cquirrel:

```
SELECT l_orderkey, o_orderdate, o_shippriority,
SUM(l_extendedprice * (1- l_discount)) AS revenue
FROM Lineitem, Customer, Orders
WHERE c_mktsegment = 'AUTOMOBILE'
AND c_custkey = o_custkey AND l_orderkey = o_orderkey
AND o_orderdate < date '1995-03-13'
AND l_shipdate > date '1995-03-13'
GROUP BY l_orderkey, o_orderdate, o_shippriority
```

The query includes a three-way join between relation Lineitem, Orders and Customer relation. Figure 2 gives a dummy database consisting of these three relations. We next demonstrate how to maintain the index structure to handle continuous updates for the query Q.

Query Plan. In Cquirrel, the relations are joined as depicted by the foreign-key DAG, which is akin to a query plan. Figure 2 shows the foreign-key DAG of TPC-H query 3 and Figure 3 shows the query plan of query 3 in Flink. For the foreign-key DAG, a vertex represents a relation, and there is a directed edge from R_i to R_j if R_i has a foreign key referencing the primary key of R_j. The directed edge is annotated with the join key. In addition to the joins, the query plan also includes two additional parts, I/O, and aggregation.



Lineitem

L_ORDERKEY	L_EP	L_DIS	L_SHIPDATE
1	21168.23	0.04	1995-03-13
1	45983.16	0.09	1995-04-12
2	13309.60	0.1	1995-01-29
2	28955.64	0.09	1995-04-21
3	22824.48	0.1	1995-03-30
3	49620.16	0.07	1995-01-30

Orders

O_ORDERKEY	O_CUSTKEY	O_ORDERDATE	O_SP
1	1	1995-01-02	1
2	2	1995-12-01	2
3	3	1995-10-14	1
4	1	1995-07-30	3
5	2	1995-02-21	4
6	3	1995-01-10	5

Customer

C_CUSTKEY	C_MKTSEGMENT
1	AUTOMOBILE
2	AUTOMOBILE
4	BUILDING
5	MACHINEY
6	AUTOMOBILE

3	AUTOMOBILE

I(Lineitem)

L_ORDERKEY	L_EP	L_DIS	L_SHIPDATE
1	21168.23	0.04	1995-03-14
	45983.16	0.09	1995-04-12
2	13309.60	0.1	1995-01-29
	28955.64	0.09	1995-04-21
3	22824.48	0.1	1995-03-30
	49620.16	0.07	1995-01-30

I$_c$(Lineitem)

L_ORDERKEY	S(t)
1	1
2	0
3	~~0~~->1
4	0
5	1
6	~~0~~->1

I(Orders)

O_CUSTKEY	O_ORDERKEY	O_ORDERDATE	O_SP
1	1	1995-01-02	1
	4	1995-07-30	3
2	2	1995-12-01	2
	5	1995-02-21	4
3	3	1995-01-14	1
	6	1995-01-10	5

I$_c$(Orders)

O_CUSTKEY	S(t)
1	1
2	1
3	~~0~~->1
4	0
5	0
6	1

I(Customer)

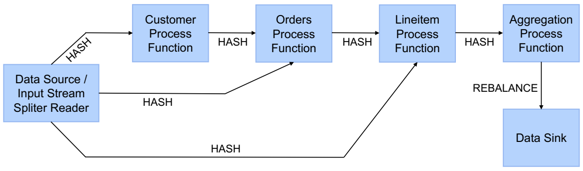
C_CUSTKEY	C_MKTSEGMENT
1	AUTOMOBILE
2	AUTOMOBILE
4	BUILDING
5	MACHINEY
6	AUTOMOBILE

3	AUTOMOBILE

I(Aggregation)

L_ORDERKEY	O_ORDERDATE	O_SP	Revenue
1	1995-01-02	1	62166.18
3	1995-01-14	1	20542.03

Figure 2: Maintaining TPC-H Query 3 in Cquirrel. Left: Dummy database instance; Middle: Foreign-key Graph; Right: Index for each relation and aggregation. Tuple in red color: alive tuples. Tuple in white color: non-live tuples. Tuple in green color: the insert tuple. Tuple in pink color: the update tuples. All unrelated attributes are omitted from the figure.



Data Source / Input Stream Spliter Reader
HASH
Customer Process Function
HASH
Orders Process Function
HASH
Lineitem Process Function
HASH
Aggregation Process Function
REBALANCE
Data Sink
HASH
HASH

Figure 3: Query Plan in Flink

Live Tuples. For each relation, Cquirrel divides all inserted tuples into two parts, *alive* and *non-live*. For any relation R, a tuple is alive if it can join with one tuple in every relation in $\mathcal{D}(R)$, where $\mathcal{D}(R)$ represents all R's descendant relations. All tuples in the leaf relation, i.e., has no descendant relation, are alive (subject to the applicable selection condition). In Figure 2, all alive tuples are marked in red and non-live tuples in white.

Another important observation is that if a tuple t is *alive* on R, then there exists an *alive* tuple in each child relation of R that can join with t. For each $R_c \in C(R)$, if such tuple exists, we said that t is alive on R_c. By using this property, Cquirrel is able to maintain all live state in a bottom-up fashion. For example, if a tuple $(3, \text{AUTOMOBILE})$ is inserted into relation Customer, then the tuple $(3, 3, 1995\text{-}01\text{-}14, 1)$ and $(6, 3, 1995\text{-}01\text{-}10, 5)$ become alive. The changing of state also triggers a bottom-up update, which makes tuple $(3, 22824.48, 0.1, 1995\text{-}03\text{-}30)$ becomes alive. In figure 2, the tuple in green is the newly inserted tuple, and all tuples that changes state during the bottom-up updates are marked in pink.

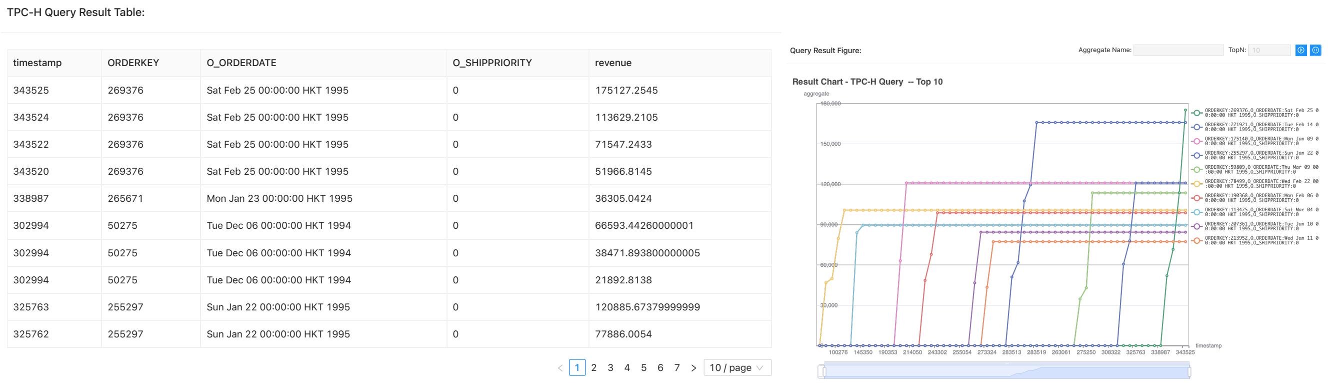
However, being alive on all $R_c \in C(R)$ is only a necessary condition for a tuple being alive. The condition is only sufficient if the foreign-key graph is a directed tree. Otherwise, the mismatching problem might arise. To solve the problem, Cquirrel introduces the concept of assertion keys as well as auxiliary attributes. By appending assertion keys and auxiliary attributes to the index, Cquirrel can detect the mismatching and mark those tuples as non-alive. Due to space constraints, we omit the details of assertion keys. The details can be found in the paper [14].

Data Structure. In Cquirrel, the system needs to maintain an index structure (see Figure 2 right) to keep track of all live tuples. The index should be able to: (1) enumerate all tuples in relation R in constant time per tuple; (2) given any key value v, enumerate all tuples whose value on key x_k is v with a constant delay, or report that there is none; (3) insert or delete a tuple in constant time; and (4) use $O(|R|)$ memory. A standard implementation of such an index is a hash table on all the distinct key values of x_k. Each slot of the hash table stores a pointer to the list of all tuples whose value on x_k is the given value. Using universal hash functions, all the above operations can be done in expected $O(1)$ time [9].

In the actual implementation, Cquirrel takes advantage of the Flink keyed streams. In Flink, a keyed stream will be partitioned by a given key. In the implementation, Cquirrel partitions a relation R by its foreign key. If a tuple $t \in R$ is alive with respect to one of its foreign key k_f, then all tuples $t' \in R$ that has same foreign key k_f as t are alive with respect to k_f. Thus, for each relation, we only need to maintain a hash set I for each foreign key k_f to store all tuples with respect to k_f, which satisfies all the requirements.

In addition, the system also maintains a counter I_c for each foreign key k_f. If the counter equals the number of child relations of R, then the foreign key k_f is alive.

Aggregation. In Cquirrel, the system plugs the multi-way joins into the delta propagation framework to handle more complex queries. For example on TPC-H query 3, Cquirrel will store a state of the aggregation value ‘revenue‘ on the current instance. As shown in [14], we can enumerate the deltas of the multi-way joins with a constant delay. When the tuple $(3, \text{AUTOMOBILE})$ is inserted, Cquirrel first updates all index for all relations, and produces a delta $(3, 22824.48, 0.1, 1995\text{-}01\text{-}14, 1)$. The aggregation state will receive all deltas and calculate the new aggregation value, which is 20542.03 for group $(3, 1995\text{-}01\text{-}14, 1)$.



TPC-H Query Result Table:

timestamp	ORDERKEY	O_ORDERDATE	O_SHIPPRIORITY	revenue
343525	269376	Sat Feb 25 00:00:00 HKT 1995	0	175127.2545
343524	269376	Sat Feb 25 00:00:00 HKT 1995	0	113629.2105
343522	269376	Sat Feb 25 00:00:00 HKT 1995	0	71547.2433
343520	269376	Sat Feb 25 00:00:00 HKT 1995	0	51966.8145
338987	265671	Mon Jan 23 00:00:00 HKT 1995	0	36305.0424
302994	50275	Tue Dec 06 00:00:00 HKT 1994	0	66593.44260000001
302994	50275	Tue Dec 06 00:00:00 HKT 1994	0	38471.893800000005
302994	50275	Tue Dec 06 00:00:00 HKT 1994	0	21892.8138
325763	255297	Sun Jan 22 00:00:00 HKT 1995	0	120885.67379999999
325762	255297	Sun Jan 22 00:00:00 HKT 1995	0	77886.0054

< 1 2 3 4 5 6 7 >
10 / page
Query Result Figure:
Aggregate Name:
TopN: 10
Result Chart - TPC-H Query -- Top 10
aggregate
180,000
150,000
120,000
90,000
60,000
30,000
timestamp
100276
145350
190353
214050
243302
255054
273324
283513
283519
263061
275250
308322
325763
338987
343525
ORDERKEY:269376,O_ORDERDATE:Sat Feb 25 00:00:00 HKT 1995,O_SHIPPRIORITY:0
ORDERKEY:221921,O_ORDERDATE:Tue Feb 14 00:00:00 HKT 1995,O_SHIPPRIORITY:0
ORDERKEY:175140,O_ORDERDATE:Mon Jan 09 00:00:00 HKT 1995,O_SHIPPRIORITY:0
ORDERKEY:255297,O_ORDERDATE:Sun Jan 22 00:00:00 HKT 1995,O_SHIPPRIORITY:0
ORDERKEY:59809,O_ORDERDATE:Thu Mar 09 00:00:00 HKT 1995,O_SHIPPRIORITY:0
ORDERKEY:78499,O_ORDERDATE:Wed Feb 22 00:00:00 HKT 1995,O_SHIPPRIORITY:0
ORDERKEY:190368,O_ORDERDATE:Mon Feb 06 00:00:00 HKT 1995,O_SHIPPRIORITY:0
ORDERKEY:113475,O_ORDERDATE:Sat Mar 04 00:00:00 HKT 1995,O_SHIPPRIORITY:0
ORDERKEY:207361,O_ORDERDATE:Tue Jan 10 00:00:00 HKT 1995,O_SHIPPRIORITY:0
ORDERKEY:213952,O_ORDERDATE:Wed Jan 11 00:00:00 HKT 1995,O_SHIPPRIORITY:0

(a) Table View

(b) Query result visualization

Figure 4: Result Visualization Component

General SQL Queries. Cquirrel can support selection, projection, and ring aggregations (for example, SUM aggregation and COUNT aggregation) in constant delay based on the delta propagation framework. By adjusting the index structure, Cquirrel can also evaluate operators like union (∪) and difference (−) between two acyclic multi-way join queries Q and Q', and semiring aggregations like MIN/MAX. More details can be found in the paper [14].

3 USER INTERFACE

The user interface contains three main components: Query input, Code generation, and Result display. Users can specify parameters in the setting tab before starting the demonstration, for example, the server address, the update sequences, or the input source.

The demonstration has four steps. First, the user will submit a SQL query or a JSON file on the Query input component. If the user submit a SQL query, the query will be sent to the SQL parser and generate the corresponding JSON file. The JSON file contains the information about the query plan. Experience users can also write the JSON file and submit the file directly to the Web UI. By manually writing the JSON file, users can submit a better query plan than the parser provided.

For the second step, the JSON file will be sent to the code generator, which is running in the background of the server. The Web UI will capture the generation log and the query plan from the code generator, and display them on the web page. The code generator will analyze the JSON and generate a corresponding Flink program. After this, the code generator will compile the generated code and generate a Jar package. The Web UI will then receive the Jar package and submit it as a job to the Flink server as the third step.

While the Flink job is running, the Web UI will receive the output records and display the result in the Result display component (see Figure 4) as the final step. In this demonstration, we integrate two different views to visualize the update stream. The first one is the table view (see Figure 4a), which contains all tuples from the output streams generated by Cquirrel; the second one is a dynamic figure (see Figure 4b). The dynamic figure will show the change of the aggregate value. It can support the TOP operator for group-by aggregation, i.e., the dynamic figure will only show the groups that have the largest (or smallest) aggregation values. For queries with multiple aggregations, the user can specify an aggregation that s/he wants to be monitored.

ACKNOWLEDGMENTS

This work has been supported by HKRGC under grants 16202317, 16201318, 16201819, and 16205420, and by Alibaba Group through the Alibaba Innovative Research Program.

REFERENCES

[1] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. Join Synopses for Approximate Query Answering. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99)*. 275–286.

[2] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *Proceedings of the VLDB Endowment* 5, 10 (2012), 968–979.

[3] Rafi Ahmed, Randall Bello, Andrew Witkowski, and Praveen Kumar. 2020. Automated generation of materialized views in Oracle. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3046–3058.

[4] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*. Springer, 208–222.

[5] Randall G Bello, Karl Dias, Alan Downing, James Feenan, Jim Finnerty, William D Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. 1998. Materialized views in Oracle. In *VLDB*, Vol. 98. 24–27.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 38, 4 (2015), 28–38.

[7] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.

[8] Rada Chirkova and Jun Yang. 2012. Materialized views. *Foundations and Trends® in Databases* 4, 4 (2012), 295–405.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.

[10] Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1259–1274.

[11] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation Reuse in Analytics Job Service at Microsoft. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 191–203.

[12] Feifei Li. 2019. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2263–2272.

[13] Stackexchange. 2021. . https://dba.stackexchange.com/questions/102903/is-it-acceptable-to-have-circular-foreign-key-references-how-to-avoid-them

[14] Qichen Wang and Ke Yi. 2020. Maintaining Acyclic Foreign-Key Joins under Updates. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD '20)*. 1225–1239.