CrocodileDB in Action: Resource-Efficient Query Execution by Exploiting Time Slackness

Dixin Tang
University of Chicago
totemtang@uchicago.edu

Zechao Shang
University of Chicago
zcshang@cs.uchicago.edu

Aaron J. Elmore
University of Chicago
aelmore@cs.uchicago.edu

Sanjay Krishnan
University of Chicago
skr@cs.uchicago.edu

Michael J. Franklin
University of Chicago
mjfranklin@uchicago.edu

ABSTRACT

Existing stream processing and continuous query processing systems eagerly maintain standing queries by consuming all available resources to finish the jobs at hand, which can be a major source of wasting CPU cycles and memory resources. However, users sometimes do not need to see the up-to-date query result right after the data is ready, and thus allow a slackness of time before the result is returned, which provides new opportunities to avoid wasting resources. We proposed CrocodileDB, a resource-efficient database, where users specify a performance goal representing the maximally allowed slackness of time and the system generates a query plan to minimize resource consumption (e.g. memory consumption or CPU cycles) while meeting this performance goal at the same time. In this paper, we demonstrate how users interact with CrocodileDB and show how the time slackness enables our optimization of reducing CPU consumption: Incrementability-aware Query Processing (InQP). With the slackness specified by users, InQP can reduce computing resource waste by selectively deferring the execution of parts of a query that are not amenable to incremental executions (i.e. outputting tuples that can be deleted by later executions in a high probability). In this demonstration, users can set the performance goal as a trade-off between CPU consumption and query latency, and observe the CPU usages and other statistics to understand how InQP reduces computing resources.

PVLDB Reference Format:
Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, Michael J. Franklin. CrocodileDB in Action: Resource-Efficient Query Execution by Exploiting Time Slackness. *PVLDB*, 13(12): 2937-2940, 2020.
DOI: https://doi.org/10.14778/3415478.3415513

1. INTRODUCTION

Resource efficiency is a crucial challenge for database designs as (1) the growth of data is outpacing the expansion of computing and memory resources, (2) environmental concerns demand more judicious use of available resources, and (3) there are emerging resource-constrained scenarios that require more efficiently utilizing limited resources for desired performance. Examples of such resource-constrained scenarios include edge computing in IoT devices with limited battery capacity and cloud-based databases with a pay-per-use-model where users want to best utilize their resources and dollars.

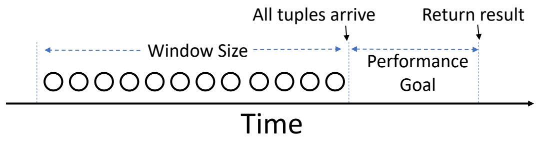


Figure 1: An example of a performance goal in CrocodileDB

Unfortunately, existing database systems for dynamic data, such as stream processing and continuous query processing [3], are not optimized for resource-efficient query execution. They eagerly maintain standing queries to immediately process new tuples by consuming all available resources. This eager query execution could significantly waste both CPU cycles and memory resources on tuples that will be removed later, and the system may excessively maintain intermediate states that are barely used for future query processing. We find in resource-constrained scenarios, users commonly do not need to see the query result immediately after the data is complete and allow a slackness of time before the result is returned. This slackness provides new opportunities for reducing resource waste, which are not fully exploited by existing systems.

We proposed CrocodileDB [5], a resource-efficient database that exploits the slackness in generating query results to minimize resource consumption. CrocodileDB integrates users' performance goals (i.e. the maximal slackness) into query planning such that the query execution plan can intelligently delay some parts of a query execution to reduce CPU consumption and discard some intermediate states for reducing memory consumption.

To enable these optimizations, we allow users to specify a *performance goal* that represents the maximally allowed time to return the result after the data is complete. Figure 1 shows an example of querying a window of data. Here, the performance goal is the maximally allowed time between when the last tuple arrives for this window and the query result is returned. Our query optimizer internally leverages the information about users' performance goals along with information about the query structure and data arrival patterns (i.e. which relations having new data and the corresponding data arrival rates) to generate a query plan that can reduce CPU consumption [7] and memory usage [6] while meeting the performance goal.

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415513

In this demonstration, we focus on our optimization technique Incrementability-aware Query Processing (InQP) [7], which prioritizes selectively maintaining parts of a standing query to reduce CPU consumption. Maintaining a standing query involves incremental execution, where new data is incrementally incorporated into prior results to reduce the time of providing up-to-date results. Incremental executions can waste CPU cycles because for some queries, tuples output in prior executions are removed by later executions. For example, consider a SQL query that finds all customers with an above-average balance. To incrementally maintain this query, on each new customer tuple, we not only need to update the running average but also re-scan all the existing customers (or search appropriate indices) to update the query result when the average balance changes. In this case, outputting the whole query result upon the arrival of each tuple wastes resources. In InQP, we define a metric, *incrementability*, to quantify how amenable a query is to incrementally executions. We further observe that a query includes substructures, each of which has a different level of incrementability. Consider the aforementioned example. Maintaining the average balance is incrementable, while outputting the average balance to maintain the query result is less incrementable. To reduce the wasted CPU cycles of incremental executions, InQP selectively executes the parts of the query with lower incrementability more lazily. For the parts of the query with higher incrementability, we execute them more eagerly to meet users' performance goals.

We note that this optimization is impossible in existing systems [1, 2, 4] because these systems, instead of allowing users to specify performance goals about when the result is desired, require users dictate when the query should be maintained (or executed), such as using time-based or count-based triggers. For example, in prior work [1, 2, 4] users need to specify a time-based trigger to dictate the refresh frequency of the whole query (e.g. update the query for every 1 min of new data). This prohibits any optimization opportunity of leveraging the different levels of incrementability within the substructures of a query to reduce computing resources. By contrast, CrocodileDB allows users to specify a performance goal and the system decides the query execution plan (e.g. when to maintain a query) to reduce resource consumption.

2. PERFORMANCE GOALS & RESOURCES

CrocodileDB maintains a standing window query over a stream of tuples. While CrocodileDB currently supports tumbling windows, we can support other window semantics, such as sliding windows. We later briefly discuss how to support performance goals in more general cases.

Performance goals for tumbling window: In CrocodileDB, users can explicitly express a performance goal, which is the maximally allowed time slackness between when all tuples for a window arrive and the actual result is returned to users.

The performance goal is a knob that users can tune to make tradeoffs between resource consumption and query latency. With different performance goals, the system will generate corresponding plans to minimize resource consumption. Consider an example of a windowed query with a window of 10 minutes. If users allow a large slackness (e.g. a performance goal of 2 mins), CrocodileDB can selectively maintain some parts of the query lazily to reduce CPU consumption [7] or selectively discard some intermediate states of incremental executions to reduce memory consumption [6]. If the slackness is large enough (e.g. 10 mins), CrocodileDB can start the query after all tuples arrive (i.e. batch processing) and avoid the CPU or memory resources waste introduced by incremental executions. On the other hand, if users prioritize query performance (e.g. return the result within 1 sec for every 10 mins of data), CrocodileDB will execute this query more eagerly with higher resource consumption. As shown in our demonstration plan of Section 4, users can observe the estimated total resource consumption, such as the total number of CPU seconds the query will use for a given performance goal.

With the performance goal specified by users, CrocodileDB unlocks many optimization opportunities [5] that are impossible in existing systems[1, 2, 4]. Existing systems let users decide when to execute the query, instead of allowing users to specify when to expect a query result in CrocodileDB. For example, users need to set a time trigger of maintaining the whole query periodically (e.g. every 1 min) to achieve the desired performance. This query plan executes the whole query in a single pace and ignores that some parts of a query are less amenable to incremental executions. In this paper, we focus on the optimization of selectively delaying some parts of a query to reduce CPU consumption, which is discussed in Section 3.

Extensions of performance goals to more general cases: The performance goal of CrocodileDB can be extended to sliding windows. Semantically, a sliding window can be regarded as a list of independent windows. We can apply the performance goal to each of them. We note that the underlying system optimizations should consider the overlaps between sliding windows to reduce redundant work, which is beyond the scope of this paper.

The performance goal can also be applied to general incremental view maintenance. Consider an example of maintaining a view over a stream of tuples. Users can specify the condition of computing an up-to-date result (e.g. updating the result for every 10 mins of data) and additionally submit a performance goal to decide when they can see an up-to-date result. For example, if the performance goal is 10 secs, for every 10 mins of data, we will incorporate them into the query result within 10 secs after the data is ready.

3. INCREMENTABILITY-AWARE QUERY PROCESSING

We now discuss how the time slackness specified by users enables a new optimization technique Incrementability-aware Query Processing (InQP [7]). InQP is optimized to minimize CPU consumption while meeting a performance goal. To meet a performance goal, the system may start query execution before all data arrive for a window and incrementally incorporate new data into prior results. Incremental executions can waste CPU cycles because tuples generated by earlier executions can be removed by later executions. Interestingly, we find that the amount of work wasted in incremental executions depends on the structures of the query and the new data. Therefore, we define a metric, *incrementability*, to quantify the cost-effectiveness of an incremental execution. The higher incrementability a query is, the less work the query will waste and thus is more amenable to incremental executions. We further observe that a query plan can be decomposed into smaller pieces (i.e. a query path in InQP) and each query path has a different level of incrementability. Intuitively, InQP will execute the query path with a higher incrementability more frequently and delay the execution of query path with a lower incrementability to reduce CPU consumption.

3.1 System Background

Given a performance goal, CrocodileDB needs to find a plan that minimizes the total resources (i.e. CPU consumption in InQP) and has a query latency (i.e. the time of returning the result after all tuples for a window arrive) no larger than the performance goal.

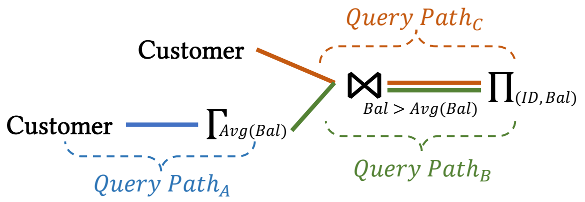


Figure 2: A query with multiple query paths

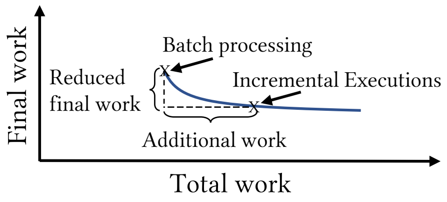


Figure 3: An example of the benefit (i.e. reduced final work) and cost (i.e. additional work) for an incremental execution plan.

In InQP, we estimate the CPU consumption and the query latency using the metrics of *total work* and the *final work* respectively. The total work represents the total amount of work done by this query based on the cost model of our existing work [7]. The final work represents the amount of work the query needs to do after all data arrives. Therefore, the goal of InQP is to minimize the total work under a given final work constraint.

A query path is a dataflow segment in the query plan delineated by blocking operators, inputs, or outputs. Note that an operator may belong to more than one query path. Figure 2 shows a example query that finds the IDs and balance of customers with a balance larger than the average balance (i.e. $Bal > Avg(Bal)$). This query has three query paths: (1) the first query path A takes balance from `Customer` to compute the average balance (i.e. $\Gamma_{Avg(Bal)}$), (2) the second query path B takes $\Gamma_{Avg(Bal)}$, joins it with the all tuples from `Customer` and outputs customer IDs and balance, and (3) query path C takes tuples from `Customer` and joins them with the average value. One execution of a query path represents flushing the buffered tuples of a blocking operator or an input relation all the way to the end of this query path where it reaches another blocking operator or the output. All blocking operators including aggregate, sort, and distinct have buffers to delay outputing new tuples. Similarly, we consider all base relations, delta logs and the final output as buffers. On the other hand, simple operators like a filter or a join output tuples without buffers.

A pace determines how frequently we execute a query path (i.e. flushing its input buffers). We assume that new tuples arrive at a steady rate and consider one flushing with respect to the percentage of the total number of tuples arrived for a time window. Each query path with a pace k will flush its input buffer whenever the system has received new $\frac{1}{k}$ of all the estimated tuples. A *pace configuration* can be represented as a vector $P = (K_1, K_2, \ldots, K_Q)$ for Q query paths. A special pace configuration $P_{\mathbb{1}} = (1, 1, \ldots, 1)$ represents the case of all tuples being processed by a single final step.

3.2 Incrementability Definition

We now define the metric of incrementability. If the system starts the query execution when all tuples arrive, the query execution has the lowest total work and the highest final work, which is the case of batch processing. Its pace configuration is $P_{\mathbb{1}}$. If we increase any paces in the pace configuration $P_{\mathbb{1}}$, we include more incremental executions, which may increase the overall total work but decrease the final work. In other words, the benefit of performing more incremental executions is the reduced final work and the corresponding cost is the increased total work. Figure 3 shows an example of the "benefit" and "overhead" of incremental executions. We define incrementability of a pace configuration P with respect to $P_{\mathbb{1}}$ as the ratio between the "benefit" (i.e. the amount of reduced final work) and the "cost" (i.e. the amount of additional total work). Note that the definition of incrementability can be generalized to any two pace configurations P_1 and P_2, where P_2 executes more eagerly than P_1. This means that each query path's pace in P_2 is no smaller than the pace in P_1, and there is at least one query path in P_2 whose pace is larger than the pace in P_1.

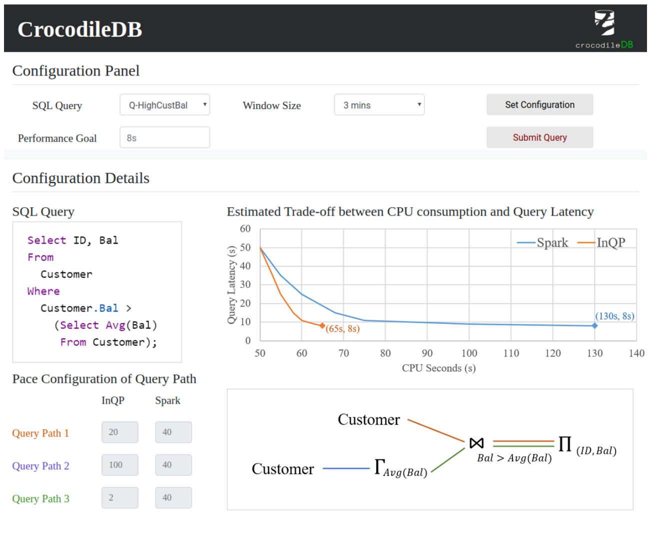
Depending on the structures of a query path and the input data, a query has different levels of incrementability when we increase paces of different query paths. If the incrementability is no larger than zero (i.e. no "benefit"), this query path is non-incrementable (i.e. increasing pace for this query path does not reduce final work). On the other hand, if the "overhead" is zero and the incrementability is ∞, this query path is fully incrementable. All cases that lie between the two ends are partially incrementable. The value of incrementability is larger than 0 but less than ∞.

3.3 Optimization via Incrementability

We use the example in Figure 2 to illustrate how to find a pace configuration that minimizes the total work and meets a final work constraint. We have three query paths in Figure 2 and assume a pace configuration $P = (P_A, P_B, P_C)$. The optimization starts with $P_{\mathbb{1}}$ (i.e. batch processing). At each optimization step, we increment the pace for one query path by 1. For the first step, we consider increasing the pace 1 to 2 for one of the three query paths. Specifically, we have three possible pace configurations to consider: $(1, 1, 2)$, $(1, 2, 1)$, and $(2, 1, 1)$. We note that $(1, 2, 1)$ is not valid since query path B is the parent of query path A and cannot execute more eagerly than query path A. We compute the incrementability of $(1, 1, 2)$ and $(2, 1, 1)$ with respect to $(1, 1, 1)$ respectively and choose to increase the pace of the path with the higher incrementability. We repeat this step by increasing one pace in the new pace configuration by 1 until we meet the final work constraint.

4. DEMONSTRATION PLAN

We implement CrocodileDB in Spark by extending Spark to support delete and update operations. We develop a framework to demonstrate how users interact with CrocodileDB and how its underlying optimization InQP can reduce CPU consumption compared to Spark with the same performance goal, where Spark uses a single uniform pace. We find this pace based on InQP's cost model to minimize the CPU consumption with respect to a performance goal. This framework contains an interactive configuration interface and a real-time performance monitoring component. The configuration interface allows users to 1) submit a window query and specify a performance goal; 2) tune the performance goal to observe the trade-off between CPU consumption and query latency (i.e. the time of returning the result after all tuples for a window arrive); and 3) observe the different pace configurations between InQP and Spark. The monitoring component allows users to compare two key metrics between InQP and Spark: 1) the CPU usages during query processing and 2) the statistics of how many prior output tuples are removed by later executions. The two metrics help users understand how the pace configuration of InQP can reduce CPU consumption by intelligently delaying the executions of query paths with lower levels of incrementability.


CrocodileDB

Configuration Panel

SQL Query: Q-HighCustBal
Window Size: 3 mins
Performance Goal: 8s

Configuration Details

SQL Query

```
Select ID, Bal
From
  Customer
Where
  Customer.Bal >
    (Select Avg(Bal)
     From Customer);
```

Pace Configuration of Query Path

	InQP	Spark
Query Path 1	20	40
Query Path 2	100	40
Query Path 3	2	40

Figure 4: CrocodileDB configuration component

4.1 Configuration

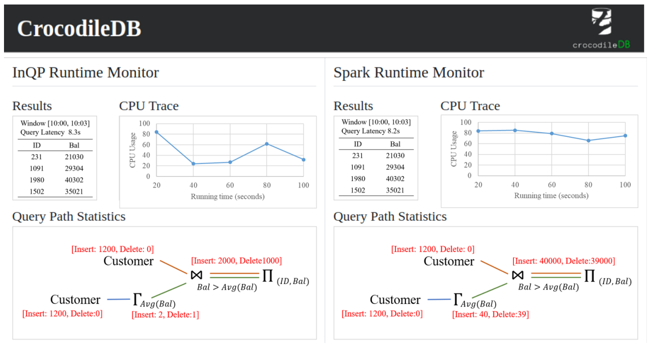
Users will experience querying a stream of tuples that are being loaded from an external source. Figure 4 shows the configuration interface of CrocodileDB. In the *Configuration Panel*, users first choose a query from all TPC-H queries and several hand-written queries based on the TPC-H schema. Then, users set the window size and the performance goal. For example, Figure 4 shows that users select *Q-HighCustBal* query to be executed. It will compute a result over every 3 mins of loaded tuples given the window size of 3 mins. For a window of tuples, the system needs to return the up-to-date result within 8 seconds after all tuples arrive, which is bounded by the performance goal. When users hit the *Set Configuration* button, the details of the configuration are shown below.

The top-left part of *Configuration Details* shows the SQL query selected by users. Users are able to observe the estimated trade-off between CPU consumption and query latency for both InQP and Spark. Given the performance goal set by users, the estimated CPU consumption is also highlighted in the trade-off curve. For example, Figure 4 shows that for a performance goal of 8 seconds, InQP is estimated to use 65 seconds of CPU time, while Spark is estimated to use 130 seconds of CPU time. Below that, users can observe the different pace configurations for both InQP and Spark, and how InQP decomposes a query plan into query paths. Each query path in the query plan has a different color and the corresponding label (e.g. *Query Path 1*) shares the same color as its query path.

If users hit the *Submit Query* button, the configuration framework will submit the query to systems InQP and Spark. Users are able to observe the runtime statistics of both systems side-by-side in our monitoring component.

4.2 Runtime Monitoring

Our monitoring component, shown in Figure 5, monitors the execution of the same query with the same performance goal for InQP and Spark side-by-side. We show the returned result and the actual latency of returning this result at the top-left corner of each system's panel. Users are expected to observe similar latencies for both systems since they use the same performance goal. To the right of the result, we have the CPU usages during the query execution. For InQP, users can observe that the system has varied CPU usages over time, where a lower CPU usage indicates the time when InQP delays the executions for some query paths. By


CrocodileDB

InQP Runtime Monitor

Results

Window [10:00, 10:03]
Query Latency 8.3s

ID	Bal
231	21030
1091	29304
1980	40302
1502	35021

Spark Runtime Monitor

Results

Window [10:00, 10:03]
Query Latency 8.2s

ID	Bal
231	21030
1091	29304
1980	40302
1502	35021

Figure 5: CrocodileDB monitoring component

contrast, Spark should have steady CPU usages over time. Users will see that InQP has lower overall CPU consumption than Spark.

Below the CPU trace, the query paths statistics show how many insert/delete tuples are output from each operator at runtime. The more delete tuples there are, the more prior output tuples are removed by later executions, which wastes CPU cycles. For example, in Figure 5 we see that the join operator in InQP outputs much less delete tuples compared to Spark. Users can compare the number of tuples output from each operator at runtime and understand why InQP can significantly reduce CPU consumption.

5. CONCLUSION

This demonstration highlights how CrocodileDB enables resource-efficient query execution. We show how users can express the maximally allowed time slackness (i.e. performance goal) and how this slackness enables the new optimization of InQP. We not only show the resource efficiency of InQP in terms of CPU consumption, but also explain the rationals behind InQP via interactive configuration interfaces and showing runtime statistics.

6. REFERENCES

[1] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[2] E. Begoli, T. Akidau, F. Hueske, J. Hyde, K. Knight, and K. Knowles. One SQL to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables. In *SIGMOD 2019*, pages 1757–1772. ACM, 2019.

[3] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD 2000*, pages 379–390, 2000.

[4] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting multiple view maintenance policies. In *SIGMOD 1997*, pages 405–416, 1997.

[5] Z. Shang, X. Liang, D. Tang, C. Ding, A. J. Elmore, S. Krishnan, and M. J. Franklin. Crocodiledb: Efficient database execution through intelligent deferment. In *CIDR 2020*. www.cidrdb.org, 2020.

[6] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Intermittent query processing. *PVLDB*, 12(11):1427–1441, 2019.

[7] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Thrifty query execution via incrementability. In *SIGMOD 2020*, pages 1241–1256. ACM, 2020.