CrocodileDB: Efficient Database Execution through Intelligent Deferment

Zechao Shang[1], Xi Liang[1], Dixin Tang[1], Cong Ding[2],
Aaron J. Elmore[1], Sanjay Krishnan[1], Michael J. Franklin[1]
[1] *University of Chicago* [2] *Peking University*
{zcshang, xiliang, totemtang, aelmore, skr, mjfranklin}@uchicago.edu
congding@pku.edu.cn

ABSTRACT

The end of Moore's law will push database system designers to be more judicious with computation as the growth in data outpaces the availability of computational resources. Eagerness, or aggressively consuming resources to immediately and quickly complete the task at hand, is one source of wasted resources in modern data systems where the systems expend unnecessary resources waiting on queries, data, or both. Intelligently deferring a task to a later point in time can increase result reuse, reduce work that might later be invalidated, or avoid unnecessary work altogether. We propose a research prototype system, CrocodileDB, which is a resource-efficient database system that automatically optimizes deferment based on user-specification and workload prediction. CrocodileDB integrates new ways of specifying timing information, new query execution policies, new task schedulers, and new data loading schemes.

1. INTRODUCTION

Reducing the consumption of resources in data systems will be crucial as Moore's law comes to an end [14]. This trend will push system designers to be more judicious with computation due to the ever-growing imbalance of data compared to available computation. Furthermore, the growing adoption of IoT and autonomous vehicles will also push systems to conserve computation as battery-backed hardware will need to trade-off performance for battery life [1]. The consolidation of commercial services, along with pay-per-use serverless architectures, and zero-carbon temporary compute resources, demands that new systems can utilize constrained and dynamic physical resources [24, 46].

Today's database systems are designed to be *eager*, where they consume resources "right away" to quickly complete a task at hand. In a traditional RDBMS, the system waits until a query arrives, and then immediately does the necessary work to fully answer the query. Likewise, streaming systems and modern incremental view maintenance (IVM) systems wait for new data to arrive, and then immediately maintain results for standing queries, derived views, and any intermediate states used to facilitate the update process [15]. Eagerness has its pitfalls: in a traditional DBMS, any work is simply redone if an identical or similar query arrives in the near future, and in a standing query system, the system holds on to all of the operator state regardless of whether new data actually arrives. Eagerness without regard to the arrival frequency and pattern of data and queries can lead to a significant amount of wasted systems resources.

While the eager design paradigm is beneficial for applications on dedicated hardware whose primary concern is low latency, we believe that modern Cloud and IoT trends encourage rethinking this approach. We consider scenarios where data is not static, but is not frequent enough to call for a streaming solution. Here, we envision an application with either a low, bursty, or periodic data ingest, or an application that may not need a frequent refresh on standing queries or views with dynamic data. Consider the following examples: an analyst or system who only needs to see updated results every 30 minutes; standing queries that are for monitoring purposes and only want to see updated results when they change beyond a given threshold (i.e. average latency $>$ 200 ms); or extract, transform, and load tasks scheduled at night, with changes from this new data not be needed until the markets open.

Modern systems, however, do not allow the application to specify this slack [7] nor do they optimize query execution or maintain views with this information in mind. To make this concrete, consider the example above where a user requires an up-to-date query result only every 30 minutes. A streaming system will hold on to all the resources needed to maintain the result for the entire 30 minutes, while a batch system will incur a high latency because it would recompute the result from scratch each interval. Ideally, a system should be able to find a middle ground by partially computing the query every so often in the 30 minute window–optimized by a planner that is given a resource utilization constraint and answers the query with the lowest apparent latency to the user. Similarly, in the ETL example, if the system has prior knowledge about the ETL schedule and knows that its data is static during those intervals, it can change its query processing to aggressively share computation by persisting common subqueries since it knows they will not require maintenance until the next ETL cycle.

While the case for avoiding eager computation (or sharing computation) has been explored in shared query execution [30, 47, 42, 40, 44, 22, 36, 20], materialized view maintenance [15], in data loading for Hadoop-style systems [2], and for databases that support web-services [19, 20, 13], "laziness" in data processing has not seen widespread adoption. One main reason is that the aforementioned concerns have slowly emerged in the last decade. Second, it is hard for applications to specify timing in SQL and current triggers [7]. Third, asynchronous, or lazy, systems can be harder to develop and debug. It is important to note that our notion of "laziness" is on of temporal deferment and this is different from the concept of lazy evaluation in data flow systems, where a chain of operations is not started until a certain operation is triggered to fold in operations and avoid data movement. The goal of our work is less about single query optimization but more about reducing global resource utilization by manipulating when queries and subqueries are executed and how their execution artifacts can be shared.

Therefore, it is our position that it is critical for a new class of systems that consider timing information of both how data arrives and how data is used in order to (a) intelligently persist the results of work to save on future computation and (b) intelligently defer query execution, view maintenance, and data loading when possible to save on computation and memory usage. Such systems should be able to collect or predict future data arrival patterns and allow for users to specify timing preferences or update policies for standing queries/views. The data arrival patterns should be exploited for determining how queries and views updates are scheduled and should be able to assess the cost of such updates. This opens up several interesting optimization questions, such as given a query and some model of future data, when is it the ideal time to begin processing the query, which is largely influenced by how incremental can the query incorporate new data. Additionally, such models can determine what results, both as sub-query results and intermediate state, should be persisted to improve future queries or updates. For long-standing queries, it may be beneficial to actively update the results intermittently and avoid wasting resources. For standing queries and views, it also is important to asses how new data will impact existing result. Understanding this without parsing all of the new tuples requires new client/server interfaces and the use of machine learning models.

To address these issues we propose *CrocodileDB*[1], a resource efficient database system that integrates timing information, policies, schedulers, data loading, view selection, and query execution. This system calls for several new system policies, planner components, execution strategies, client interfaces, and data storage strategies. Section 2 outlines the high-level architecture of CrocodileDB and introduces major components. Section 3 discusses related work relevant to our system. Section 4 details key components, previews initial results, and discusses relevant related work for specific components.

2. SYSTEM OVERVIEW

Figure 1 outlines the key components of CrocodileDB that we describe in this section. Several of these components are further detailed in Section 4. We are currently building our prototype using a modified version of SparkSQL, but none of these components are predicated on Spark specific concepts. The figure's top tier highlights policy and planning

[1]Crocodiles frequently lay still to conserve energy and move quickly at the last possible moment.

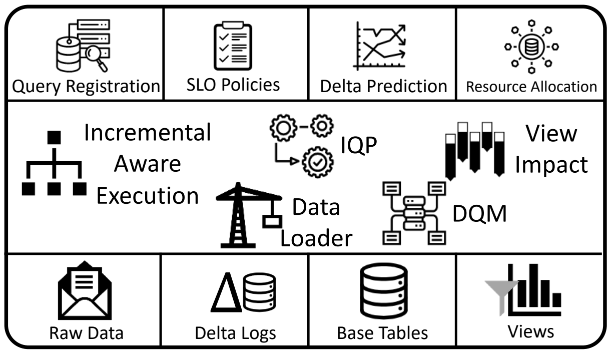

Figure 1: CrocodileDB overview.

components, the middle tier shows view and query execution components, and the bottom tier shows the physical data organization. For exposition, assume there is a database with new records coming in at fixed intervals, with occasional deletes and updates to existing records, and several teams of data scientists register long-standing queries that have results updated (or refreshed) every 30 minutes and many materialized views are created that are updated every 30 minutes or when the view result has an aggregated record whose answer changes by more than 10%. Data is organized into (a) raw data that has been received by clients, but has not been processed and remains in "wire format"; (b) delta logs, which are the parsed data from the client that specifies either the new records or updates/deletes to existing records; (c) base tables that have records previously integrated from prior delta logs (stored in Parquet for our prototype); and (d) materialized views and pre-generated results (stored in Parquet). The execution layer contains five modules, in addition to standard query execution components.

IAE: *Incremental Aware Execution (IAE)* analyzes queries to understand how amenable they are for incremental computation and uses cost models of "incrementability" to control scheduling of queries and control flow of complex long-standing queries. In our motivating example, IAE would determine when a query refresh should begin based on how incremental the work is. For example, queries that are amenable to incremental computation can begin early, otherwise, it might be beneficial to defer. IAE relies on the *Query Registration* and *SLO policies* to determine when to schedule the query. In addition, a *Delta Prediction* component informs IAE where data changes are likely to occur. IAE can reduce CPU cycles and memory consumption, by avoiding potentially unnecessary work.

IQP: *Intermittent Query Processing (IQP)* allows for long-standing queries to be refreshed intermittently, with minimizing resource utilization (e.g. memory) in-between execution intervals. In our example, after each refresh, the user is given new tuples or changes from prior results, and until the next execution IQP selectively persists state (i.e. hash tables or materialized operators) from query execution to enable low-latency refreshes. IQP relies on the delta prediction component for understanding how the base data will change, the *resource allocation* component to provide resources to maintain the query while inactive to determine the ideal states to keep given the allocated resources and delta prediction. IQP can reduce memory required for maintaining standing queries and can reduce CPU cycles for integrating new data, by selectively maintaining state from query execution.

View Impact: As CrocodileDB allows for view maintenance to be deferred until a result changes by a threshold according to registered queries and SLO policies, the *view impact* monitors new raw data and delta logs to determine which existing views could be invalidated. This relies on a formal reasoning framework to see how a set of tuples with bounded values will change results: producing a certificate of optimality or unboundedness. This calculation is performed without fully integrating the new or updated records into the base tables and subsequently, the materialized views. View Impact can reduce CPU cycles and memory consumption, by avoiding updating standing queries or views if the change is within the user's accepted bounds.

DQM: Given that resources, such as memory and computation, are being freed from lazy execution a natural question to explore is how can the free memory be used to accelerate query processing through opportunistic view creation as a side-effect of query execution. *Deep Q-Materialization (DQM)* explores how to take extra memory given by the resource allocation and knowledge of where data is likely to change from the delta predictor, to determine which views would benefit query latency overall for the system's workload. DQM can reduce CPU cycles, by using excess memory to cache frequent sub-results.

Data Loader: To aid components that need to understand data before it is fully loaded, such as IAE and View Impact, CrocodileDB relies on a *Data Loader* component to quickly parse, analyze, potentially defer the loading of new data before it goes through an expensive data loading process [18, 5]. To improve the ability to understand new data and partially load new data [2], the data loader extends the client interface to specify how data should be organized [41] and pushes limited query predicates to the client (i.e. indicate the max value from a block of records). The Data Loader can reduce CPU cycles and memory usage, by avoiding potentially expensive parsing and validation through partial deferred loading.

Language Extensions: To exploit timing information for lazy execution, CrocodileDB will need to extend SQL to allow users to specify when and how results should be updated [7]. This could be given as the following for materialized views:

```
CREATE MATERIALIZED VIEW view1 AS
SELECT ...
REFRESH WHEN [interval] or [condition]
```

or for intermittent queries as:

```
INTERMITTENT [how]
SELECT ...
INTERVAL [when] TERMINATE ON [condition]
```

3. RELATED WORK

Data analytics systems can be characterized by two architectural extremes. On one side, they are optimized for ingesting data as efficiently as possible and executing analytics in a batch-oriented way [2, 17] (i.e., optimized for write-throughput). On the other extreme, they can maintain complex intermediate states to be able to answer standing queries or queries on materialized views as efficiently as possible in a streaming setting [29, 21, 28, 11, 38] (i.e., optimized for read-latency). The primary knob for systems designers to span the middle ground between these extremes is varying the amount of materialization [8, 15, 20].

This simplified trade-off space assumes that the user desires results that are exact (fresh) as of the point at which they issued the query. If no assumptions are made about when queries can be issued, the system has to ensure that all derived states are consistent and ready to be read at all times [15]. In many applications, this notion of freshness is unnecessary–a user might only care if their answer is consistent with respect to a snapshot that is no more than 30 minutes old. If the execution engine knew this information upfront it could significantly optimize the maintenance of a desired standing query or a materialized view. The "slack" time specified by the user could increase result reuse [20], reduce work that might later be invalidated (e.g., maintaining non-monotonic queries) [15], or avoid unnecessary work altogether (e.g., a later task deletes all the records) [13].

Such optimizations are only possible if we decouple timing from query processing and data ingest and allow the user to specify softer timing constraints. Timing in data systems, namely, when analytics or update tasks are launched, is taken as given—governed by the arrival of queries (e.g., traditional relational databases), data (e.g., continuous query systems [6]), or both (e.g., PSoup [12]). To the best of our knowledge, this is an architectural insight not considered thoroughly in prior work. While laziness has been considered before, lazy view maintenance [50] is a form of deferred materialized view maintenance [16] that defers updates until the data is queried. Similarly, related insights are found in the query scheduling literature [9] and in the systems scheduling community [48]. We build CrocodileDB with timeliness as a first-class citizen. We study how known timing semantics can be exploited at operator-level, view level, and workload-level to reduce wasted work.

Deferred execution enables opportunities of other query optimization approaches, including but not limited to shared query execution. Concurrently executed queries could share a part of their executions, for example, table scans and joins. Although sharing execution may compromise some queries' performance, it reduces overall execution time (or memory consumption). Existing literature of shared execution examines the query to be executed by two queries, and find identical or overlapping workloads to be shared. CrocodileDB is different from shared query execution in that it systematically combines the knowledge about new data, query semantics, and users' expectation together to reduce overall query processing cost. A seminal work of shared query execution is QPipe [22]. It defines several query evaluation stages, where each stage corresponds to a type of operator (e.g. Join and Aggregate) and is assigned a dedicated pool of threads. A new query is decomposed into the predefined stages and routed among stages based on its query plan. Each stage exploits the opportunities of sharing work of concurrently running operators such as reusing a hash table for a hash join operator. SharedDB and BatchDB [36, 20] delay query execution, batch several queries, and build a single query execution plan to maximally share the work of batched queries. Several other works consider specific operators or applications such as sharing scan [42, 40, 44] and joins [35] in main-memory databases, and sharing the query execution for star schema in data warehouses [10]. This idea of shared query execution is also widely used in continuous query processing [30, 47] and adaptive query processing [34].

4. CrocodileDB COMPONENTS

In this Section we outline key components for CrocodileDB. Preliminary reports on IQP and DQM are available for the interested reader [45, 33].

4.1 Intermittent Query Processing

Query evaluation generates intermediate states, such as hash tables or materialized relations. A critical question is whether a database should keep these states if the underlying data will change. Conventional databases optimized for faster data ingestion tend to keep and maintain all states to quickly update a result. Those optimized for lower resource consumption (and less frequent result refreshes) discard all states after execution. Either way, if the data does not change in the frequency that the database expects, the system could either waste resources (e.g. memory for keeping states) or time (e.g. rebuilding states). We propose a new paradigm Intermittent Query Processing (IQP), where predictable data arrival are exploited for opportunities with database engine designs.

The core of our IQP prototype is *Delta-oriented intermediate state selection (DISS)* that selectively keeps a subset of intermediate states within a memory budget for accelerating data ingestion and query result refreshment based on a predictive model of future data [45]. DISS considers three types of intermediate states according to the query plan that initially processes the data: 1) states that are generated and materialized (temporarily) in the query plan; 2) states that are generated, but not materialized in the query plan (e.g. output tuples for pipeline operators); 3) states that are not generated by the query plan. All intermediate states incur a space (memory) cost for materialization, and the latter two types of states incur an additional computation cost of materializing or generating them. We build a dynamic programming algorithm to holistically consider which states to keep/materialize/build. Its optimization goal is to minimize the (expected) time of incorporating new data plus the time of materializing new states under a given memory budget.

We implement an initial DISS prototype on PostgreSQL, and compare it with two approaches: ReBatch that does not keep intermediate state and recompute all the time, and DBToaster that eagerly maintain intermediate states. We port DBToaster's plan on PostgreSQL and denote it as DBT-PG. We use the join ordering benchmark (JOB) [32]. that includes 21 IMDB tables (4.3 GB data in total) and 33 queries that involve from 4-way to 16-way join. Our experiments assume that 99% of data is present in the first execution and 1% of data is the "delta". If a query cannot finish within 500s, we mark it as DNF. Here, we assume the memory budget is sufficient for all systems.

Figure 2 shows the total processing time, and that DISS is faster than ReBatch and DBT-PG. We also report the memory consumption ratio between DBT-PG and DISS in Figure 3. It shows that DISS consumes much less memory than DBT-PG. For future work on IQP, we plan to explore how to shrink memory budgets over time as the predicted delta size shrinks, how to allocate memory resources across multiple queries, and how to re-optimize a query plan (i.e. different join orders) for quickly processing deltas.

4.2 Incremental Aware Execution

For CrocodileDB to efficiently defer execution of queries, it is critical to understand how incrementable a query is. As certain queries are more amenable to incremental computation, the system could efficiently start refreshing a result before it is needed [49]. For example consider the following two queries (1) select `sum(val) where val > 10`, and (2) `select sum(val) where val > avg(val)`. The first query is highly incrementable, and as soon as new records are received they can be integrated to the result without much maintenance. However, for the second query if new records are expected, working on the query early may result in wasted work as new records can change `avg(val)`, invalidating prior records in the result or requiring new tuples to be added. We informally define incrementability as how much of the work that could currently be done, will likely contribute to the final result. To demonstrate how incrementability impacts performance consider the results in Figure 4. We divide a TPC-H dataset into a different number of batches, incrementally execute them, and report the ratio between the total execution time of different number of batches and the time of executing the full dataset as one batch. Here we use a modified version of Spark that eliminates the start-up costs of each incremental execution. As Figure 4 shows, if the ratio is close to 1, it means the query is incrementable, and we prefer to execute this query more eagerly. Otherwise, increasing the number of executions introduces a much higher total execution time (e.g. Q15).

This component will build a cost model to understand how incremental a query actually is [3, 23]. Here, given a query, it will capture what the trade-offs are for batch computation and incremental computation. With this understanding, we will then explore how to integrate the delta predictions to augment this incremental model. This will allow the system to make decisions about how to process a delta (e.g. delay until the batch is complete, one tuple at a time, or start at some point in the interval). This requires understanding how inserts, updates, and deletes to base tables will impact a query's execution. This opens interesting questions on how to optimize queries that are not amenable for incremental execution. We plan to also explore how to improve query execution by exploiting knowledge of how amenable it is for incremental execution [49].

4.3 View Impact: Accuracy-based Specs

CrocodileDB decouples the scheduling of update and query tasks from data arrivals or query arrival. So far, we have discussed this tolerance in terms of temporal conditions, e.g., a query result based on a snapshot that is no more than 30 minutes old. An intriguing extension is to consider accuracy-based specifications, where updates are processed only if they significantly impact a standing query.

CrocodileDB contains a framework to estimate how aggregates queries could change given buffered delta data. This estimation is only useful if computing change estimate is significantly cheaper than actually running the query itself. While sampling has been previously used to estimate changes in materialized views [31], we believe that the probabilistic nature of sampling guarantees makes it challenging to operationalize. For nightly updates to a database, a sampled estimate would expect to exceed a 3 standard deviation confidence interval once a year. We would like hard guarantees based on coarse-grained statistics collected from the delta data.

We propose an axiomatic method for summarizing relational data called a Predicate Constraint (PC). A PC models

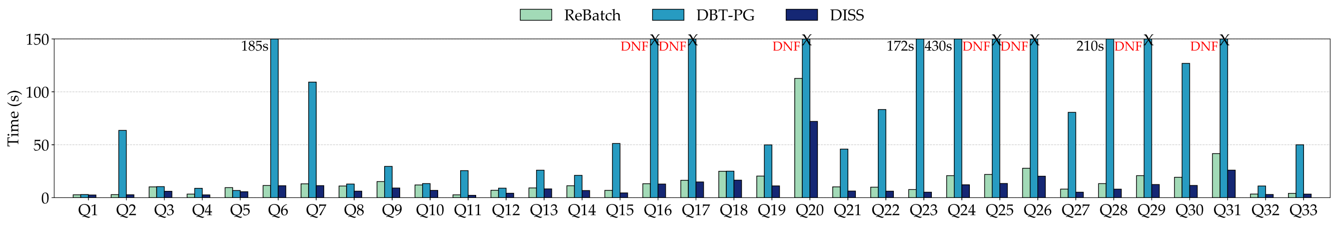

Figure 2: Total processing time (initial query and a single 1% delta) with the join ordering benchmark.

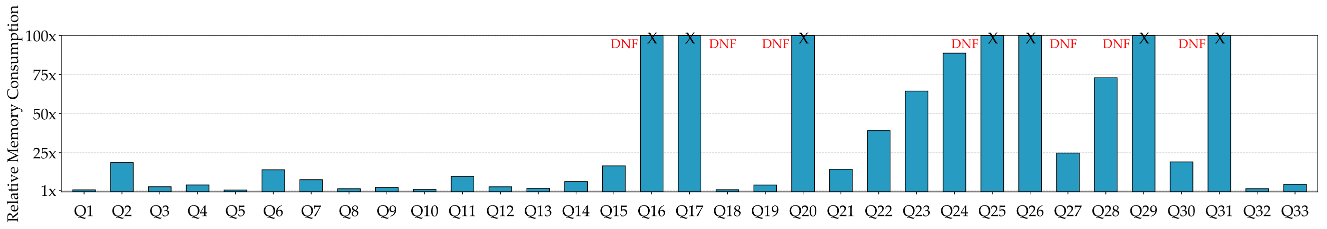

Figure 3: Relative memory consumption between DBT-PG and DISS on the join order benchmark.

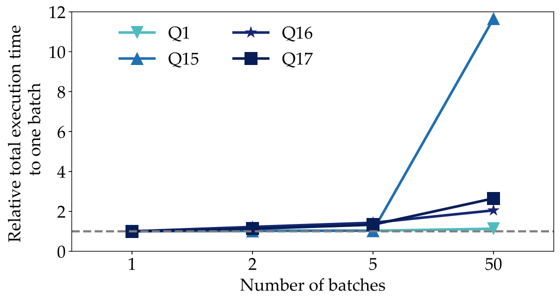

Figure 4: Evaluating the impact of incrementability on selected TPC-H queries on a modified Spark.

"what" happens at "'where". Each PC contains two parts: a *predicate* and a *constraint*. The predicate specifies a subdomain of data. It models "where". The constraint bounds a summarization of the change that may happen in this subdomain, or in other words, it models "what". This summarization may specify how many tuples will change at most, and the magnitude of the change on attributes. We generate sets of such constraints to summarize a delta table. If a potential change, for example a missing tuple, satisfies multiple predicates, it is under multiple corresponding constraints. We estimate the potential impact on the query result when the change of data conforms to a set of PCs.

It is important to note, that our objective is not to build the most expressive language to represent uncertain data but rather one that we can pragmatically use to bound error in aggregate queries. PCs are more expressive in terms of representing the multiplicity of tuples than classical missing-data representations (c-tables [25] and m-tables [43]), but less expressive in terms of representing possible values that unprocessed data could take.

We estimate the impact by annotating each relation with a set of PCs, and propagate the set through relational operators. Once the base tables are evaluated to the final relation, we use its associated PC set to derive an estimation of the impact. The end-user may manually specify PCs using domain knowledge. Or, the system could also automatically generate PCs by analyzing the delta data.

4.4 DQM: Opportunistic Caching

If a system has knowledge about future updates and knows that its data is static during those intervals, it can change its query processing to share computation by persisting common subqueries since it knows they will not require maintenance until the next ETL cycle. Materialized views have the potential to greatly increase the performance of queries but it is often challenging to decide when and what view to materialize especially under constraints like limited storage. In CrocodileDB, we use deep reinforcement learning to learn adaptive view materialization and eviction policies in an opportunistic manner [33]. This means that intermediate results are cached as an artifact of execution. Opportunism couples view creation with query execution, where it might be beneficial to force a sub-optimal query plan that generates an intermediate result that is useful for future query processing—and this is what makes this optimizer different from standard view recommendation [27, 26, 4].

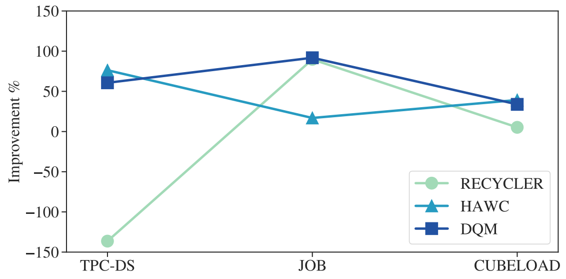

Figure 5: We compare DQM to different baselines on 3 workloads. Even including learning time, DQM outperforms or performs competitively with the best baselines on different workloads.

The system is fully adaptive and learns an effective view selection policy with an asynchronous RL algorithm that runs paired counter-factual experiments during system idle times. The optimizer contains four major components: 1) a view candidate miner that analyzes the past queries and generate possible view candidates for selection, 2) a reinforcement learning model that factors in the current system

Relevant Attrs	Full (s)	Partial (s)	Part+Split (s)
2/9	91.697	44.323	24.836
3/9	94.444	49.302	31.302
6/9	94.784	55.155	46.132

Table 1: Ingesting JSON data is expensive due to parsing and the client can reorder fields or provide splitting hints to make parsing more efficient. We consider 3 data loading scenarios with a varied number of relevant attributes and how different client optimizations can improve loading.

state, the past queries, the feedback of previous decisions and learns to decide which view to be materialized at the current time-step, 3) a credit-based eviction policy that decides which view should be evicted when there is no storage for new views and 4) a query re-writer that re-writes query with materialized views.

We evaluate the optimizer, which we call DQM, with workloads of different characteristics generated based on the Join Order Benchmark (JOB), TPC-DS and CUBELOAD, which simulates real-life usage of OLAP systems. We generated a query workload from the templates in these benchmarks by perturbing the templates with different filters and skewing their frequencies. We also compare DQM with the two state-of-the-art baselines: HAWC [39] and RECYCLER [37]. Figure 5 shows the improvement of each baseline over the cost of original workloads without using views. Overall, from a performance perspective, our experiment results show that DQM outperforms or performs competitively with the best baselines on different workloads even with learning overhead included. The micro-benchmark experiment results also suggest DQM is adaptive, robust and effective in different situations. In our future work, we plan to further improve DQM to work in a more complicated environment, for example, a system with multiple users.

4.5 Data Loading

Data loading is an expensive process due to parsing, validations, integrity checking, and data structure maintenance. To minimize this cost, prior projects explored hardware accelerators [18], minimizing data loading [2], and working directly on the raw data [5]. By borrowing from ideas in client interface redesign [41], CrocodileDB will explore adaptive transport formats to make partial data loading faster, where prior work relied on required attribute splitting for an entire record [2].

These projects influence CrocodileDB to include a lightweight and just-in-time data loading. Table 1 (Full) illustrates query plus loading times for loading JSON data into a Parquet file and then querying it. Table 1 (Partial) show that if the CrocodileDB knows that only certain attributes will be read, it can greatly save on parsing ($\sim 50\%$). Queries on a database, in general, have to wait until all the tuples are loaded for their results to be complete. If we had timing requirements on different queries, we could opt to load attributes relevant to one query first and defer the others to a later point in time. Furthermore, there are opportunities to further optimize loading if the client can do some processing work. We see that with the client information we can greatly reduce the time of parsing JSON if the client can reorder fields or provide information about splitting attributes.

5. CONCLUSION

We introduce CrocodileDB as a new system to exploit timing information to trade-off resource consumption and latency when possible. CrocodileDB proposes new execution and view maintenance policies that rely on policy and planner components to make informed decisions. The trade-offs in this system open up several exciting research directions.

6. REFERENCES

[1] Internet of things. https://www.gartner.com/en/information-technology/insights/internet-of-things, 2019.

[2] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible loading: access-driven data transfer from raw files into database systems. In *EDBT*, pages 1–10. ACM, 2013.

[3] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. 24(2):245–256, 1995.

[4] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, pages 496–505, 2000.

[5] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: efficient query execution on raw data files. In *SIGMOD*, pages 241–252.

[6] S. Babu and J. Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[7] 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*, pages 1757–1772. ACM, 2019.

[8] D. Borthakur et al. Apache hadoop goes realtime at facebook. In *SIGMOD*, pages 1071–1080. ACM, 2011.

[9] L. Bouganim, F. Fabret, C. Mohan, and P. Valduriez. Dynamic query scheduling in data integration systems. In *ICDE '00*, pages 425–434, 2000.

[10] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2(1):277–288, 2009.

[11] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *TCDE Bulletin*, 36(4), 2015.

[12] S. Chandrasekaran and M. J. Franklin. Psoup: a system for streaming queries over streaming data. In *VLDB*, pages 140–156, 2003.

[13] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). *TODS*, 2016.

[14] A. A. Chien and V. Karamcheti. Moore's law: The first ending and a new beginning. *Computer*, 46(12):48–53, 2013.

[15] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[16] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, pages 469–480, 1996.

[17] C. Doulkeridis and K. NØrvåg. A survey of large-scale

analytical query processing in mapreduce. *VLDB Journal*, 23(3):355–380, 2014.

[18] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien. UDP: a programmable accelerator for extract-transform-load workloads and more. In *MICRO*, pages 55–68, 2017.

[19] C. Garrod, A. Manjhi, A. Ailamaki, B. Maggs, T. Mowry, C. Olston, and A. Tomasic. Scalable query result caching for web applications. *PVLDB*, 1(1):550–561, 2008.

[20] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: killing one thousand queries with one stone. *PVLDB*, 5(6):526–537, 2012.

[21] J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araújo, M. Ek, E. Kohler, M. F. Kaashoek, and R. Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *OSDI*, pages 213–231, 2018.

[22] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 383–394, 2005.

[23] H. He, J. Xie, J. Yang, and H. Yu. Asymmetric batch incremental view maintenance. In *ICDE*, pages 106–117, 2005.

[24] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[25] T. Imieliński and W. Lipski Jr. Incomplete information in relational databases. In *Readings in Artificial Intelligence and Databases*, pages 342–360. Elsevier, 1989.

[26] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Selecting subexpressions to materialize at datacenter scale. *PVLDB*, 11(7):800–812, 2018.

[27] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation reuse in analytics job service at microsoft. In *SIGMOD*, pages 191–203, 2018.

[28] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In *ICDE*, pages 1507–1518, 2018.

[29] O. A. Kennedy, Y. Ahmad, and C. Koch. Agile views in a dynamic data management system. In *CIDR '11*, number CONF, pages 284–295, 2011.

[30] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 623–634, 2006.

[31] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, and T. Kraska. Stale view cleaning: Getting fresh answers from stale materialized views. *PVLDB*, 8(12):1370–1381, 2015.

[32] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 2015.

[33] X. Liang, A. J. Elmore, and S. Krishnan. Opportunistic view materialization with deep reinforcement learning. *arXiv preprint arXiv:1903.01363*, 2019.

[34] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 49–60, 2002.

[35] D. Makreshanski, G. Giannikis, G. Alonso, and D. Kossmann. Mqjoin: Efficient shared execution of main-memory joins. *PVLDB*, 9(6):480–491, 2016.

[36] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. Batchdb: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 37–50, 2017.

[37] F. Nagel, P. Boncz, and S. D. Viglas. Recycling in pipelined query evaluation. In *ICDE '13*, pages 338–349, April 2013.

[38] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: stateful scalable stream processing at linkedin. *PVLDB*, 10(12):1634–1645, 2017.

[39] L. L. Perez and C. M. Jermaine. History-aware query optimization with materialized intermediate views. In *ICDE '14*, pages 520–531, 2014.

[40] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core cpus. *PVLDB*, 1(1):610–621, 2008.

[41] M. Raasveldt and H. Mühleisen. Don't hold my data hostage: a case for client protocol redesign. *VLDB*, pages 1022–1033, 2017.

[42] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 60–69, 2008.

[43] B. Sundarmurthy, P. Koutris, W. Lang, J. Naughton, and V. Tannen. m-tables: Representing Missing Data. In M. Benedikt and G. Orsi, editors, *ICDT*, volume 68, pages 21:1–21:20, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[44] M. Switakowski, P. A. Boncz, and M. Zukowski. From cooperative scans to predictive buffer management. *PVLDB*, 5(12):1759–1770, 2012.

[45] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Intermittent Query Processing. *PVLDB*, pages 1427–1441, 2019.

[46] A. Verbitski et al. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD*, pages 1041–1052. ACM, 2017.

[47] S. Wang, E. A. Rundensteiner, S. Ganguly, and S. Bhatnagar. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 619–630, 2006.

[48] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278. ACM, 2010.

[49] K. Zeng, S. Agarwal, and I. Stoica. iolap: Managing uncertainty for efficient incremental OLAP. In *SIGMOD*, pages 1347–1361, 2016.

[50] J. Zhou, P. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *VLDB*, pages 231–242, 2007.