Crystal: A Unified Cache Storage System for Analytical Databases

Dominik Durner*
Technische Universität München
dominik.durner@tum.de

Badrish Chandramouli
Microsoft Research
badrishc@microsoft.com

Yinan Li
Microsoft Research
yinan.li@microsoft.com

ABSTRACT

Cloud analytical databases employ a disaggregated storage model, where the elastic compute layer accesses data persisted on remote cloud storage in block-oriented columnar formats. Given the high latency and low bandwidth to remote storage and the limited size of fast local storage, caching data at the compute node is important and has resulted in a renewed interest in caching for analytics. Today, each DBMS builds its own caching solution, usually based on file- or block-level LRU. In this paper, we advocate a new architecture of a smart cache storage system called *Crystal*, that is co-located with compute. Crystal's clients are DBMS-specific "data sources" with push-down predicates. Similar in spirit to a DBMS, Crystal incorporates query processing and optimization components focusing on efficient caching and serving of single-table hyper-rectangles called regions. Results show that Crystal, with a small DBMS-specific data source connector, can significantly improve query latencies on unmodified Spark and Greenplum while also saving on bandwidth from remote storage.

PVLDB Reference Format:
Dominik Durner, Badrish Chandramouli, and Yinan Li. Crystal: A Unified Cache Storage System for Analytical Databases. PVLDB, 14(11): 2432 - 2444, 2021.
doi:10.14778/3476249.3476292

1 INTRODUCTION

We are witnessing a paradigm shift of analytical database systems to the cloud, driven by its flexibility and pay-as-you-go capabilities. Such databases employ a tiered or disaggregated storage model, where the elastic *compute tier* accesses data persisted on independently scalable remote *cloud storage*, such as Amazon S3 [3] and Azure Blobs [36]. Today, nearly all big data systems including Apache Spark, Greenplum, Apache Hive, and Apache Presto support querying cloud storage directly. Cloud vendors also offer cloud services such as AWS Athena, Azure Synapse, and Google BigQuery to meet this increasingly growing demand.

Given the relatively high latency and low bandwidth to remote storage, *caching data* at the compute node has become important. As a result, we are witnessing a renewed spike in caching technology for analytics, where the hot data is kept at the compute layer in fast local storage (e.g., SSD) of limited size. Examples include the Alluxio [1] analytics accelerator, the Databricks Delta Cache [9, 15], and the Snowflake cache layer [13].

*Work done at Microsoft Research.

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476292

1.1 Challenges

These caching solutions usually operate as a black-box at the file or block level for simplicity, employing standard cache replacement policies such as LRU to manage the cache. In spite of their simplicity, these solutions have not solved several architectural and performance challenges for cloud databases:

- Every DBMS today implements its own caching layer tailored to its specific requirements, resulting in a lot of work duplication across systems, reinventing choices such as what to cache, where to cache, when to cache, and how to cache.
- Databases increasingly support analytics over raw data formats such as CSV and JSON, and row-oriented binary formats such as Apache Avro [6] – all very popular in the data lake [16]. Compared to binary columnar formats such as Apache Parquet [7], data processing on these formats is slower and results in increased costs, even when data has been cached at compute nodes. At the same time, it is expensive (and often less desirable to users) to convert all data into a binary columnar format on storage, particularly because only a small and changing fraction of data is actively used and accessed by queries.
- Cache utilization (i.e., value per cached byte) is low in existing solutions, as even one needed record or value in a page makes it necessary to retrieve and cache the entire page, wasting valuable space in the cache. This is true even for optimized columnar formats, which often build per-block *zone maps* [21, 40, 48] (min and max value per column in a block) to avoid accessing irrelevant blocks. While zone maps are cheap to maintain and potentially useful, their effectiveness at block skipping is limited by the fact that even one interesting record in a block makes it necessary to retrieve it from storage and scan for completeness.
- Recently, cloud storage systems are offering predicate push-down as a native capability, for example, AWS S3 Select [4] and Azure Query Acceleration [35]. Push-down allows us to send predicates to remote storage and avoid retrieving all blocks, but exacerbates the problem of how to leverage it for effective local caching.

1.2 Opportunities

In an effort to alleviate some of these challenges, several design trends are now becoming commonplace. Database systems such as Spark are adopting the model of a plug-in "data source" that serves as an input adapter to support data in different formats. These data sources allow the *push-down* of table-level predicates to the data source. While push-down was developed with the intention of data pruning at the source, we find that it opens up a new opportunity to leverage semantics and cache data in more efficient ways.

Moreover, there is rapid convergence in the open-source community on Apache Parquet as a columnar data format, along with highly efficient techniques to apply predicates on them using LLVM with Apache Arrow [5, 8]. This opens up the possibility of system

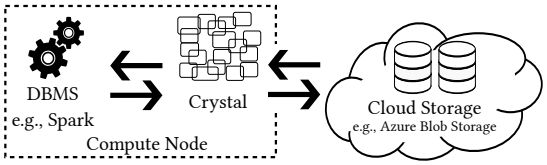


Figure 1: Crystal in big data ecosystem.

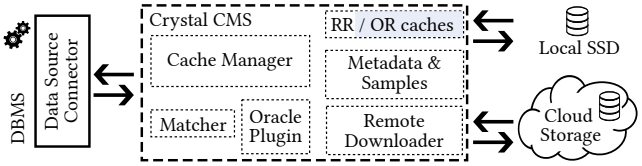


Figure 2: Crystal components.

designs that perform a limited form of data processing and transformation *outside* the core DBMS easily and without sacrificing performance. Further, because most DBMSs support Parquet, it gives us an opportunity to cache data in a DBMS-agnostic way.

1.3 Introducing Crystal

We propose a new "smart" storage middleware called *Crystal*, that is decoupled from the database and sits between the DB and raw storage. Crystal may be viewed as a mini-DBMS, or *cache management system* (CMS), for storage. It runs as two sub-components:

- The Crystal CMS runs on the compute node, accessible to local "clients" and able to interact with remote storage.
- Crystal's clients, called *connectors*, are DB-specific adapters that themselves implement the data source API with push-down predicates, similar to today's CSV and Parquet data sources.

Crystal manages fast local storage (SSD) as a cache and talks to remote storage to retrieve data as needed. Unlike traditional file caches, it determines which *regions* (parts of each table) to transform and cache locally in columnar form. Data may be cached in more than one region if necessary. Crystal receives "queries" from clients, as requests with push-down predicates. It responds with local (in cache) or remote (on storage) paths for files that cover the request. The connectors pass data to the *unmodified* DBMS for post-processing as usual. Benefits of this architecture include:

- It can be shared across multiple unmodified databases, requiring only a lightweight DBMS-specific client connector component.
- It can download and transform data into automatically chosen semantic regions in the cache, in a DBMS-agnostic columnar format, reusing new tools such as Parquet and Arrow to do so.
- It can independently optimize what data to transform, filter, and cache locally, allowing multiple views of the same data, and efficiently match and serve clients at query time.

These architectural benefits come with technical challenges (Section 2 provides a system overview) that we address in this paper:

- (Sections 2 & 3) Defining an API and protocol to communicate region requests and data between Crystal and its connector clients.
- (Section 3) Efficiently downloading and transforming data to regions in the local cache, managing cache contents, and storing meta-data for matching regions with push-down predicates over diverse data types, without impacting query latency.
- (Section 4) Optimizing the contents of the cache while: (1) balancing short-term needs (e.g., a burst of new queries) vs. long-term query history; (2) handling queries that are not identical but often overlap; (3) exploiting the benefit of duplicating frequently accessed subsets of data in more than one region; and (4) taking into account the overhead incurred by creating many small files in block columnar format, instead of fewer larger ones; and (5) managing statistics necessary for the above tasks.

Using Crystal, we get ***lower query latencies and more efficient use of the bandwidth between compute and storage***, as compared to state-of-the-art solutions. We validate this by implementing Crystal with Spark and Greenplum connectors (Section 5). Our evaluation using common workload patterns shows Crystal's ability to outperform block-based caching schemes with lower cache sizes, improve query latencies by up to 20x for individual queries (and up to 8x on average), adapt to workload changes, and save bandwidth from remote storage by up to 41% on average (Section 6).

We note that Crystal's cached regions may be considered as materialized views [20, 27, 44, 45] or semantic caches [14, 29, 30, 41–43, 47], thereby inheriting from this rich line of work. Our caches have the additional restriction that they are strictly the result of single-table predicates (due to the nature of the data source API). Specifically, Crystal's regions are disjunctions of conjunctions of predicates over each individual table. This restriction is exploited in our solutions to the technical challenges, allowing us to match better, generalize better, and search more efficiently for the best set of cached regions. As data sources mature, we expect them to push down cross-table predicates and aggregates in future, e.g., via *data-induced predicates* [28]. Such developments will require a revisit of our algorithms in future; for instance, our region definitions will need to represent cross-table predicates. We focus on read-only workloads in this paper; updates can be handled by view invalidation (easy) or refresh (more challenging), and are left as future work. Finally, we note that Crystal can naturally benefit from remote storage supporting push-down predicates; a detailed study is deferred until the technology matures to support columnar formats natively (only CSV files are supported in the current generation). We cover related work in Section 7 and conclude in Section 8.

2 SYSTEM OVERVIEW

Figure 1 shows where Crystal fits in today's cloud analytics ecosystem. Each compute node runs a DBMS instance; Crystal is co-located on the compute node and serves these DBMS instances via data source connectors. The aim is to serve as a caching layer between big data systems and cloud storage, exploiting fast local storage in compute nodes to reduce data accesses to remote storage.

2.1 Architecture

A key design goal is to make Crystal sufficiently generic so that it can be plugged into an existing big data system with minimum engineering effort. Therefore, Crystal is architected as two separate components: a light DBMS-specific data source connector and the Crystal CMS process. These are described next.

2.1.1 Data Source Connector. Modern big data systems (e.g., Spark, Hive, and Presto) provide a *data source API* to support a variety of data sources and formats. A data source receives push-down filtering and column pruning requests from the DBMS through this API.

Thus, the data source has the flexibility to leverage this additional information to reduce the amount of data that needs to be sent back to the DBMS, e.g., via block-level pruning in Parquet. In this paper, we refer to such push-down information as a *query* or *requested region*. A Crystal connector is integrated into the unmodified DBMS through this data source API. It is treated as another data source from the perspective of the DBMS, and as a client issuing queries from the perspective of the Crystal CMS.

2.1.2 Crystal CMS. Figure 2 shows the Crystal CMS in detail. It maintains two local caches – a small *requested region* (RR) cache and a large *oracle region* (OR) cache – corresponding to short- and long-term knowledge respectively. Both caches store data in an efficient columnar open format such as Parquet. Crystal receives "queries" from connectors via the Crystal API. A query consists of a request for a file (remote path) with push-down predicates. Crystal first checks with the `Matcher` to see if it can cover the query using one or more cached regions. If yes (cache hit), it returns a set of file paths from local storage. If not (cache miss), there are two options:

(1) It responds with the remote path so that the connector can process it as usual. Crystal optionally requests the connector to store the downloaded and filtered region in its RR cache.
(2) It downloads the data from remote, applies predicates, stores the result in the RR cache, and returns this path to the connector.

Thus, the RR cache is populated *eagerly* by either Crystal or the DBMS. Not every requested region is cached eagerly; instead an LRU-2 based decision is taken per request.

More importantly, in the background, Crystal collects a historical trace of queries and invokes a caching `Oracle Plugin` module to compute the best content for the OR cache. The new content is populated using a combination of remote storage and existing content in the RR and OR caches. Section 3 covers region processing in detail, while Section 4 covers cache optimization.

2.2 Generality of the Crystal Design

As mentioned above, Crystal is architected with a view to making it easy to use with any cloud analytics system. Crystal offers three extensibility points. First, users can replace the caching oracle with a custom implementation that is tailored to their workload. Second, the remote storage adapter may be replaced to work with any cloud remote storage. Third, a custom connector may be implemented for each DBMS that needs to use Crystal.

The connector interfaces with Crystal with a generic protocol based simply on file paths. Cached regions are stored in an open format (Parquet) rather than the internal format of a specific DBMS, making it DBMS-agnostic. Further, a connector can feed the cached region to the DBMS by simply invoking its built-in data source for the open format (e.g., the built-in Parquet reader in Spark) to read the region. Thus, the connector developer does not need to manually implement the conversion, making its implementation a fairly straightforward process. In Section 5, we discuss our connectors for Spark and Greenplum, which take less than 350 lines of code.

2.3 Revisiting the Caching Problem

Leveraging push-down predicates, Crystal caches different subsets of data called regions. Regions can be considered as views on the table, and are a form of semantic caching [14, 29, 30, 42, 43, 47]. Compared to traditional file caching, the advantage of semantic caching is two-fold. First, it usually returns a much tighter view to the DBMS, and thus reduces the need to post-process the data, saving I/O and CPU cost. Second, regions can be much smaller than the original files, resulting in better cache space utilization and higher hit ratios. For example, Figure 3 shows a case where regions capture all views of all queries, whereas LRU-based file caching can only keep less than half of these views.

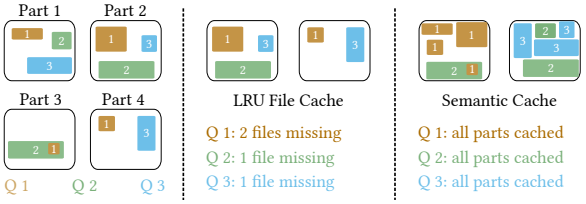


Figure 3: Benefit of semantic vs. traditional file caching. The DBMS schedules Q1, Q2, and Q3 more frequently. Only the semantic cache can answer these without remote access.

Cached regions in Crystal may overlap. In data warehouses and data lakes, it is common to see that a large number of queries access a few tables or files, making overlapping queries the norm rather than the exception at the storage layer. Therefore, Crystal has to take overlap into account when deciding which cached data should be evicted. To the best of our knowledge, previous work on replacement policies for semantic caching does not consider overlap of cached regions (see more details in Section 7).

With overlapping views, the replacement policy in Crystal becomes a very challenging optimization problem (details in Section 4). Intuitively, when deciding if a view should be evicted from the cache, all other views that are overlapping with this view should also be taken into consideration. As a result, traditional replacement policies such as LRU that evaluate each view independently are not suitable for Crystal, as we will show in the evaluation (Section 6).

Recall that we split the cache into two regions: requested region (RR) and oracle region (RR). The OR cache models and solves the above problem as an optimization problem, which aims to find the nearly optimal set of overlapping regions that should be retained in the cache. Admittedly, solving the optimization problem is expensive and thus cannot be performed on a per-request basis. Instead, the OR cache recomputes its contents periodically, and thus mainly targets queries that have sufficient statistics in history. In contrast, the RR cache is optimized for new queries, and can react immediately to workload changes. Intuitively, the RR cache serves as a "buffering" region to temporarily store the cached views for recent queries, before the OR cache collects sufficient statistics to make longer-term decisions. This approach is analogous to the C-Store architecture [46], where a writable row store is used to absorb newly updated data before it is moved to a highly optimized column store in batches. Collectively, the two regions offer an efficient and reactive solution for caching.

3 REGION PROCESSING

In this section, we focus on region matching and the creation of cached regions. Before we explain the details of the process of

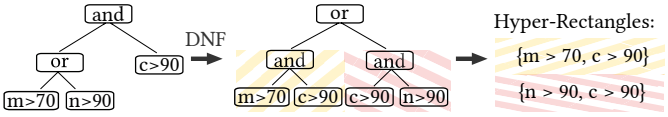


Figure 4: Transforming any predicate tree into unions of hyper-rectangles.

creating regions and matching cached regions to requests, we first show how to transform client requests into region requests.

3.1 API

Crystal acts as a storage layer of the DBMS. It runs outside the DBMS and transfers information via a minimalistic socket connection and shared space in the filesystem (e.g., SSDs, ramdisk). During a file request, the DBMS exchanges information about the file and the required region with Crystal. Because access to remote files is expensive, Crystal tries to satisfy the request with cached files.

The overall idea is that Crystal overwrites the accessed file path such that the DBMS is pointed to a local file. For redirecting queries, Crystal relies on query metadata such as the file path, push-down predicates, and accessed fields. Crystal evaluates the request and returns a cached local file or downloads the requested file. Afterward, the location of the local file is sent to the DBMS which redirects the scan to this local file. Crystal guarantees the completeness for a given set of predicates and fields. Internally, Crystal matches the query metadata with local cache metadata and returns a local file if it satisfies the requirements.

We use a tree string representation for push-down predicates in our API. Since predicates are conventionally stored as an AST in DBMS, we traverse the AST to build the string representation. Each individual item uses the syntax similar to *operation(left, right)*. We support binary operators, unary operators, and literals which are the leaf nodes of the tree. The binary operation is either a combination function of multiple predicates (such as *and*, *or*) or an atomic predicate (such as *gt*, *lt*, *eq*, ...). Atomic predicates use the same binary syntax form in which *left* represents the column identifier and *right* the compare value. To include the negation of sub-trees, our syntax allows *operation(exp)* with the operation *not*.

3.2 Transformation & Caching Granularity

Crystal receives the string of push-down predicates and transforms it back to an internal AST. Because arguing on arbitrarily nested logical expressions (with *and* and *or*) is hard, Crystal transforms the AST to Disjunctive Normal Form (DNF). In the DNF, all conjunctions are pushed down into the expression tree, and conjunctions and disjunctions are no longer interleaved. In Crystal, regions are identified by their disjunction of conjunctions of predicates. Regions also contain their sources (i.e., the remote files) and the projection of the schema. This allows us to easily evaluate equality, superset, and intersection between regions which we show in Section 3.3.

The construction of the DNF follows two steps. First, all negations are pushed as far as possible into the tree which results in Negation Normal Form (NNF). Besides using the De-Morgan rules to push down negations, Crystal pushes the negations inside the predicates. For example, *not(lt(id, 1))* will be changed to *gteq(id, 1)*.

After receiving the NNF, Crystal distributes conjunctions over disjunctions. The distributive law pushes *or*s higher up in the tree which results in the DNF. It transforms *and(a, or(b, c))* to *or(and(a, b), and(a, c))*. Although this algorithm could create 2^n leaves in theory, none of our experiments indicate issues with blow-up.

Because the tree is in DNF, the regions store the pushed-down conjunctions as a list of column restrictions. These conjunctions of restrictions can be seen as individual geometric hyper-rectangles. Regions are fully described by the disjunction of these hyper-rectangles. Figure 4 shows the process of creating the DNF and extracting the individual hyper-rectangles. Although we use the term hyper-rectangles, the restrictions can have different shapes. Crystal supports restrictions, such as *noteq*, *isNull*, and *isNotNull*, that are conceptually different from hyper-rectangles.

Crystal's base granularity of items is on the level of regions, thus all requests are represented by a disjunction of conjunctions. However, individual conjunctions of different regions can be combined to satisfy an incoming region request. Some previous work on semantic caching (e.g., [14, 17]) considers only non-overlapping hyper-rectangles. Non-overlapping regions can help reduce the complexity of the decision-making process. Although this is desirable, non-overlapping regions impose additional constraints.

Splitting the requests into sets of non-overlapping regions is expensive. In particular, the number of non-overlapping hyper-rectangles grows combinatorial. To demonstrate this issue, we evaluated three random queries in the lineitem space which we artificially restrict to 8 dimensions [23]. If we use these three random hyper-rectangles as input, 16 hyper-rectangles are needed to store all data non-overlapping. This issue arises from the number of dimensions that allow for multiple intersections of hyper-rectangles. Each intersection requires the split of the rectangle. In the worst case, this grows combinatorial in the number of hyper-rectangles.

Because all extracted regions need statistics during the cache optimization phase, the sampling of this increased number of regions is not practical. Further, the runtime of the caching policies is increased due to the larger input which leads to outdated caches.

Moreover, smaller regions require that more cached files are returned to the client. Figure 5 shows that each additional region incurs a linear overhead of roughly 50ms in Spark. The preliminary experiment demonstrates that splitting is infeasible due to the combinatorial growth of non-overlapping regions. Therefore, Crystal does not impose restrictions on the semantic regions themselves. This raises an additional challenge during the optimization phase of the oracle region cache, which we address in Section 4.5.

3.3 Region Matching

With the disjunction of conjunctions, Crystal determines the relation between different regions. Crystal detects equality, superset, intersections, and partial supersets relations. Partial supersets contain a non-empty number of conjunctions fully.

Crystal uses intersections and supersets of conjunctions to argue about regions. Conjunctions contain restrictions that specify the limits of a column. Every conjunction has exactly one restriction for each predicated column. Restrictions are described by their column identifier, their range (*min, max*), their potential equal value, their set of non-equal values and whether *isNull* or *isNotNull* is set. If two

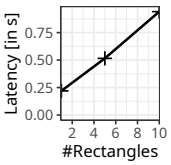


Figure 5: Many regions overhead.

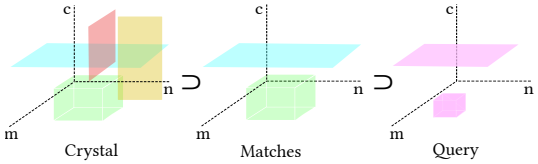


Figure 6: Crystal matches Query.

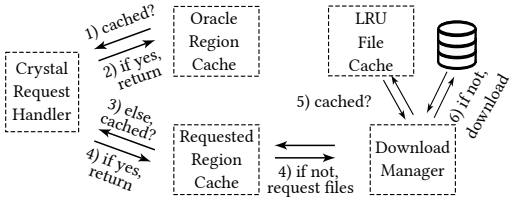


Figure 7: Request handling.

restrictions p_x and p_y are on the same column, Crystal computes if p_x completely satisfies p_y or if p_x has an intersection with p_y. For determining the superset, we first check if the null restrictions are not contradicting. Second, we test whether the (*min, max*) interval of p_x is a superset of p_y. Afterward, we check whether p_x has restricting non-equal values that discard the superset property and if all additional equal values of p_y are also included in p_x.

For two conjunctions c_x and c_y, $c_x \supset c_y$ if c_x only contains restrictions that are all less restrictive than the restrictions on the same column of c_y. Thus, c_x must have an equal number or fewer restrictions which are all satisfying the matched restrictions of c_y. Otherwise, $c_x \not\supset c_y$. c_x can have fewer restrictions because the absence of a restriction shows that the column is not predicated.

In the following, we show the algorithms to determine the relation between two regions r_x and r_y.

- $r_x \supset r_y$ holds if all conjunctions of r_y find a superset in r_x.
- $r_x \cap r_y \neq \emptyset$ holds if at least one conjunction of r_x finds an intersecting conjunction of r_y.
- $\exists\, \text{conj} \subset r_x : \text{conj} \subset r_y$ (partial superset) holds if at least one conjunctions of r_y finds a superset in r_x.
- $r_x = r_y$: $r_x \supset r_y \wedge r_y \supset r_x$

Figure 6 shows an example that matches a query that consists of two hyper-rectangles to two of the stored regions.

3.4 Request Matching

During region requests, Crystal searches the caches to retrieve a local superset. Figure 7 shows the process of matching the request. First, the oracle region cache is scanned for matches. If the request is not fully cached, Crystal tries to match it with the requested region cache. If the query was not matched, the download manager fetches the remote files (optionally from a file cache).

During the matching, a full superset is prioritized. Only if no full superset is found, Crystal tries to satisfy the individual conjunctions. The potential overlap of multiple regions and the overhead shown in Section 3.2 are the reasons to prefer full supersets. If an overlap is detected between A and B, Crystal needs to create a reduced temporary file. Otherwise, tuples are contained more than once which would lead to incorrect results. For example, it could return A and $B - A$ to the client. The greedy algorithm, presented in Algorithm 1 reduces the number of regions if multiple choices are possible. We choose the region that satisfies most of the currently unsatisfied conjunctions and continue until all have been satisfied.

We optimize the matching of regions by partitioning the cache according to the remote file names and the projected schema. The file names are represented as (bit-)set of the remote file catalog. This set is sharded by the tables. Similarly, the schema can be represented as a (bit-)set. The partitioning is done in multiple stages. After the fast file name superset check, all resulting candidates are tested for a superset of the schema. Only within this partition of superset regions, we scan for a potential match. Although no performance issues arise during region matching, multi-dimensional indexes (e.g., R-trees) can be used to further accelerate lookups.

Algorithm 1: Greedy reduction of multiple matches

```
input  :Region requestedRegion, List<Regions> partialMatches
output:List<Regions> regions
BitSet<requestedRegion.disjunctionCount> matches(0);
while true do
    if matches.isAllBitsSet() then
        return regions
    bestRegion = {}; bestVal = 0
    foreach p ∈ partialMatches do
        curval = additionalMatches(p, matches)
        if curVal > bestVal then
            bestRegion = p; bestVal = curVal
    if !bestRegion then return {}
    partialMatches = partialMatches \ bestRegion
    regions = regions ∪ buildTempFile(bestRegion, regions)
    matches.setAll(requestedRegion.satisfiedConjunctions(bestRegion))
```

3.5 Creating Regions

The cached regions of Crystal are stored as Apache Parquet files. Crystal leverages Apache Arrow for reading and writing snappy encoded Parquet files. Internally, Parquet is transformed into Arrow tables before Crystal creates the semantic regions.

Gandiva, which is a newly developed execution engine for Arrow, uses LLVM compiled code to filter Arrow tables [8]. As this promises superior performance in comparison to executing tuple-at-a-time filters, Crystal translates its restrictions to Gandiva filters. When Crystal builds new Parquet files to cache, the filters are compiled to LLVM and executed on the in-memory Arrow data. Afterward, the file is written to disk as snappy compressed Parquet file. If a file is accessed the first time, Crystal creates a sample that is used to predict region sizes and to speed up the client's query planning.

3.6 Client Database Connector

Database systems are often able to access data from different formats and storage layers. Many systems implement a connection layer that is used as an interface between the DBMS and the different formats. For example, Spark uses such an abstraction layer - known as data source.

Crystal is connected to the DBMS by implementing such a small data source connector. As DBMSs can process Parquet files already, we can easily adapt this connector for Crystal. Crystal interacts with the DBMS via a socket connection and transfers files via shared disk space or ramdisk. Since Crystal returns Parquet files, the DBMS can already process them without any code modifications.

The only additional implementation needed is the exchange of control messages. These consist of only three different messages and the responses of Crystal. One of the messages is optional and is used to speed up query planning. The scan request message and the message that indicates that a scan has finished are required by all Crystal clients. The first message includes the path of the remote file, the push-down predicates, and the required fields of the schema. Crystal replies with a collection of files that can be used instead of the original remote file. The finish message is required to delete cached files safely that are no longer accessed by the client. The optional message inquires a sample of the original data to prevent storage accesses during query planning.

3.7 Cloud Connection

Crystal itself also has an interface similar to the data source. This interface is used to communicate with various cloud connectors. The interface implements simple file operations, such as listings of directories and accesses to files. For blob storage, the later operation basically downloads the file from remote storage to the local node.

Recently, cloud providers have been adding predicate push-down capabilities to their storage APIs, e.g., S3 Select [4]. Clients can push down filters to storage and receive the predicated subset. This feature can incur additional monetary costs, as well as a per-request latency. Crystal complements this feature naturally, as it is aware of semantic regions and can use predicate push-down to populate its cache efficiently. As Crystal can reuse cached results locally, it can save on future push-down costs as well.

Crystal implements a download manager that fetches blobs from remote and stores them into ramdisk. The client is pointed to this location, and as soon as it finishes accessing it, the file is deleted again. Multiple accesses can be shared by reference counting.

4 CACHE OPTIMIZATION

This section summarizes the architecture of our caches, followed by more details on caching. Finally, we explain our algorithms that explore and augment the overlapping search space.

4.1 Requested Region and Oracle Region Cache

Recall that Crystal relies on two region caches to capture short- and long-term trends. The *RR* cache is an eager cache that stores the result of recently processed regions. The long-term insights of the query workload are captured by the *OR* cache. This cache leverages the history of region requests to compute the ideal set of regions to cache locally for best performance. Crystal allows users to plug-in a custom oracle; we provide a default oracle based on a variant of Knapsack (covered later). After the oracle determines a new set of regions to cache, Crystal computes these regions in the background and updates the *OR* cache. The creation in the background allows to schedule more expensive algorithms (runtime) to gather meaningful insights. This allows for computing (near-) optimal results and the usage of machine learning in future work. The oracle runs in low priority, consuming as little CPU as possible during high load.

An interesting opportunity emerges from the collaboration between the two caches. If the *OR* cache decided on a set of long-term relevant regions, the requested region cache does not need to compute any subset of the already cached long-term regions. On the other hand, if the requested region cache has regions that are considered for long-term usage, the *OR* cache can take control over these regions and simply move them to the new cache.

4.2 Metadata Management

A key component for predicting cached regions is the history of requested regions. To recognize patterns, the previously accessed regions are stored within Crystal. We use a ring-buffer to keep the most recent history. Each buffer element represents a single historic region request which has been computed by a collection of (remote) data files. These files are associated with schema information, tuple count, and size. The selectivity of the region is captured by result statistics. The database can either provide result statistics, or Crystal will compute them. Crystal leverages previously created samples to generate result statistics. In conjunction with the associated schema information, Crystal predicts the tuple count and the result size.

4.3 Oracle Region Cache

Long-term trends are detected by using the oracle region cache. An oracle decides according to the seen history which regions need to be created. The history is further used as a source of candidate regions that are considered to be cached.

The quality of the cached items is evaluated with the recent history of regions. Each cached region is associated with a benefit value. This value is the summation of bytes that do not need to be downloaded if the region is stored on the DBMS node. In other words, how much network traffic is saved by processing the history elements locally. Further, we need to consider the costs of storing candidate regions. The costs of a region are simply given by the size it requires to be materialized. The above caching problem can be expressed as the knapsack problem: maximize $\sum_{i=1}^{n} b_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \leq W$ where $x_i \in \{0, 1\}$. The saved bandwidth by caching a region is denoted by b, the size of the materialized cache by w. If the region is picked $x = 1$, otherwise $x = 0$. The goal is to maximize the benefit while staying within the capacity W.

However, the current definition cannot capture potential overlap in regions well. As the benefit value is static, history elements that occur in multiple regions would be added more than once to the overall value. Thus the maximization would result in a suboptimal selection of regions. In Section 4.5, we show the adaptations of our proposed algorithm to compensate for the overlapping issue.

4.4 Knapsack Algorithms

Dynamic programming (DP) can be used to solve the knapsack optimally in pseudo-polynomial time. The most widespread algorithm iterates over the maximum number of considered items and the cache size to solve the knapsack optimal for each sub-problem instance. Combining the optimally solved sub-problems results in the optimal knapsack, but the algorithm lies in the complexity of $O(n * W)$. Another possible algorithm iterates over the items and benefit values, and lies in $O(n * B)$ (B denotes maximum benefit).

In our caching scenario, we face two challenges with the DP approach. First, both W (bytes needed for storing the regions) and B (bytes the cached element saves from being downloaded) are large. Relaxing these values by rounding to mega-bytes or gigabytes reduces the complexity, however, the instances are not solved

Algorithm 2: Overlap Greedy Knapsack

```
input  :List<Region> history, List<Region> candidates, Int maxCacheSize
output :List<Region> cache
List<Region> cache = List<Region>(); Int currentCacheSize = 0
Map<Float, Region> benefitRatioMap = evaluate(candidates, history, cache)
foreach {benefit, region} ∈ benefitRatioMap do
    if currentCacheSize + region.size > maxCacheSize then
        return cache
    foreach item ∈ cache do
        if item ⊆ region then
            cache = cache \ item
    cache = cache ∪ region
    benefitRatioMap = evaluate(candidates, history, cache)
    currentCacheSize += region.size
return cache
```

optimally anymore. Second, the algorithm considers that each sub-problem was solved optimally. To solve the overlapping issue, only one region is allowed to take the benefit of a single history element. An open question is to decide which sub-problem receives the benefit of an item that can be processed with several regions.

Since many knapsack instances face a large capacity W and unbound benefit B, approximation algorithms were explored. In particular, the algorithm that orders items according to the benefit-cost ratio has guaranteed bounds and a low runtime complexity of $O(n * log(n))$. The algorithm first calculates all benefit ratios $v = \frac{b}{w}$ and orders the items accordingly. In the next step, it greedily selects the items as long as there is space in the knapsack. Thus, the items with the highest cost to benefit ratio v are contained in the knapsack. This algorithm solves the relaxed problem of the fractional knapsack optimal which loosens $x \in \{0, 1\}$ to $x \in [0, 1]$ [24].

4.5 Overlap-aware Greedy Algorithm

This greedy knapsack algorithm is used as the basis of our adaptations. In contrast to DP, this approach gives us an order of the picked items which allows us to incorporate the benefit changes.

Algorithm 2 shows the adapted greedy knapsack algorithm. The general idea is that we recompute the benefit ratio for each picked item. For each iteration step, we reevaluate the benefit and size of the current candidate set. The evaluation function sorts the input according to this benefit ratio. Thus, regions that result in higher returns in comparison to the caching size are picked earlier. Note that we only consider regions that have a benefit ratio > 1 to reduce unnecessary computation for one-time requests. The runtime complexity of the adapted algorithm is $O(n^2 * log(n))$.

The evaluation of the benefit ratio is adapted according to the previously chosen regions. We define three geometric rules which change the ratio of unpicked elements.

(1) if a candidate is a superset of a picked item, we reduce the weight and the benefit by the values of the picked elements.
(2) if a candidate is a subset of an already picked item, we reduce the benefit to 0 as it does not provide any additional value.
(3) if a candidate is intersected with an already picked item, we reduce the benefit by the history elements that are covered completely by both regions.

(1) A container region $r_c = \{r_1, r_2, \ldots, r_n, r_x\}$ fully contains n stand-alone regions and the remainder region r_x. The cost of r_c is computed by $w_c = w_x + \sum_{i=1}^{n} w_i$ and the benefit $b_c = b_x + \sum_{i=1}^{n} b_i$. If a region r_k is fully contained in another region r_c, we reduce both

Algorithm 3: Approximative Merging Augmentation

```
input  :List<Region> history, Int maxRegions, Int maxSize, Int maxCacheSize
output :List<Region> resultRegions
// RegionStruct consists of Region, Quality (0), and Size Savings (0)
  List< RegionStruct<Region, Int, Int> > enlargedRegions
foreach r ∈ history do
    foreach r' ∈ history \ {r_0, ..., r} do
        r.enlargeAll(r', enlargedRegions)
foreach r ∈ enlargedRegions do
    foreach r' ∈ history do
        if r.region.satisfies(r') then
            r.quality += 1; r.sizeSavings += r'.size
sort(enlargedRegions, λ (r1, r2) { r1.quality > r2.quality })
while !enlargedRegions.empty() ∧ maxRegions > 0 do
    r = enlargedRegions.pop(); considered = true
    foreach r' ∈ resultRegions do
        if r'.satisfies(r.region) ∧ r'.size < maxSize then
            considered = false
    if !considered then
        continue
    r.region.computeStatisticsWithSample()
    if r.region.size < maxSize ∨ (r.region.size < r.sizeSavings ∧ r.region.size <
      maxCacheSize) then
        resultRegions = resultRegions ∪ r.region; maxRegions -= 1
return resultRegions
```

the weight and benefit of r_c when r_k is picked. Thereby, we simulate r_c' which is a non overlapping version of r_c with $v_k >= v_c >= v_{c'}$. In the case, the greedy algorithm picks r_c' in a future iteration, we actually add r_c and remove the previously picked item r_k.

(2) If r_c is picked, all the other included regions in r_c are fully contained with their benefits and weights. Since the greedy algorithm picks $r_c \Rightarrow \forall r \in r_c : v_c >= v_r$. The benefit of all contained r is reduced to 0 as all history elements are included in r_c.

(3) Besides full containment, regions can have partial overlap. Assume that r_x and r_y overlap partially, and r_x is picked. Our algorithm reduces the benefit b_y by all history elements that are covered by both r_x and r_y. However, we cannot reduce the costs of caching r_y as we would need to compute the non-overlapping part of the regions. This is in direct contradiction to the goal of minimizing region splits as shown in Section 3.2. For retaining optimality, all interleaving regions must be considered as the potentially picked item in an individual branch of the problem. The branch that yields the maximum benefit is chosen as the winner. Unfortunately, this introduces exponential growth of the search space. Our experiments show that even without considering all paths, our greedy algorithm produces highly effective *OR* caches. Although this revokes the fractional knapsack optimality guarantee, our greedy algorithm only picks the locally optimal choice and does not branch.

4.6 Region Augmentation

To predict regions that are accessed in the future, the oracle needs to generalize. If the candidate set of the decision-making solely consists of the seen history elements, the oracle will overfit. Thus, a crucial part is the augmentation of the candidate set to include unseen regions that are evaluated according to the seen history.

To find generalized candidate sets, we developed the approximative merging algorithm. This algorithm tries to merge intersecting regions to find the generalized region of interest. In particular, we combine two predicates and for each predicate the global min and global max are used as new dimension restrictions. As this introduces n^2 new regions, we only merge conjunctions if they intersect

in at least one dimension. To overcome the issue of non-intersecting but neighboring hyper-rectangles (e.g., $x < 1, x \geq 1$), we allow for approximative intersections that add a small delta to the boundaries.

The full approximative merging procedure is presented in Algorithm 3. First, we compute enlarged regions from the history and consider the ones that match the previously described criteria. After determining new enlarged regions, each enlarged region is assigned a quality and size saving value. Quality counts how many history regions can be processed with this enlarged region. The overall sum of the size required by each region, that can be processed with this new enlarged region, denotes the size saving. With these properties, Crystal ranks the new regions according to quality and adds the highest ranked ones to the candidate set. We only add new regions if these cannot be represented by already existing regions and their size overhead is either smaller than a defined maximum size or the size saving is larger than the region itself. The sizes of the enlarged regions are computed with the help of the samples already collected for each file. In the experimental evaluation, we add at most 20% of additional regions (according to the history size) and define a maximum size of 20% of the total semantic cache size.

4.7 Requested Region Cache

The requested region cache is similar to a traditional cache but with semantic regions instead of pages. It decides in an online fashion whether the requested region should be cached. The algorithm must be simple to reduce decision latencies. Traditional algorithms, such as LRU and its variants, are good fits in terms of accuracy and efficiency. Besides the classic LRU cache, experiments showed the benefit of caching regions after the second (k-th in general) occurrence. With the history already available for *OR*, this adaption is simple and does not introduce additional latency. For combined *OR* and *RR* with *LRU-k*, it is beneficial to reduce the history size by the *RR/OR* split as long-term effects are captured by *OR*.

One of the biggest advantages of the *RR* cache is the fast reaction to changes in the queried workload. In comparison to the *OR* cache that only refreshes periodically, the request cache is updated constantly. This eager caching, however, might result in overhead due to additional writing of the region file. To overcome this issue, the client DBMS can simultaneously work on the raw data and provide the region as a file for Crystal; this extension is left as future work.

5 IMPLEMENTATION DETAILS

Crystal is implemented as a stand-alone and highly parallel process that sits between the DBMS and blob storage. This design helps to accelerate workloads across different database systems. Crystal is a fully functional system that works with diverse data types and query predicates, and is implemented in C++ for optimal performance.

Parallel Processing within Crystal. Latency critical parts of Crystal are optimized for multiple connections. Each new connection uses a dedicated thread for building the predicate tree and matching cached files. If a file needs to be downloaded, it is retrieved by a pool of download threads to saturate the bandwidth. All operations are either implemented lock-free, optimistically, or with fine-grained shared locks. Liveness of objects and their managed cached files is tracked with smart pointers. Therefore, Crystal parallelizes well and can be used as a low latency DBMS middleware.

Crystal also handles large files since some systems do not split Parquet files into smaller chunks. During matching we recognize which parts of the original file would have been read and translate it to the corresponding region in the cached files. Further, we are able to parallelize reading and processing Parquet files.

Spark Data Source. For our evaluation, we built a data source to communicate between Spark and Crystal, by extending the existing Parquet connector of Spark with less than 350 lines of Scala code. The connector overrides the scan method of Parquet to retrieve the files suggested by Crystal. Because Spark pushes down predicates to the data source, we have all information available for using the Crystal API. As Spark usually processes one row iterator per file, we developed a meta-iterator that combines multiple file iterators transparently (Crystal may return multiple regions). The connector is packaged as a small and dynamically loaded Java jar.

Greenplum Data Source. Further, we built a connector for Greenplum which is a cloud scale PostgreSQL derivative with an external extension framework – called PXF [34, 51]. PXF allows one to access Parquet data from blob storage [52]. We modified the Parquet reader such that it automatically uses Crystal if available. Our changes to the Greenplum connector consist of less than 150 lines of code. Without recompiling the core database, Crystal accelerates Greenplum by dynamically attaching the modified PXF module.

Both connectors currently do not support sending regions back to Crystal; instead, Crystal itself handles additions to the RR cache.

Azure Cloud Connection. We use Azure Blob Storage to store remote data, using a library called `azure-storage-cpplite` [37] to implement the storage connector. The library just translates the file accesses to CURL (HTTPS) requests. Other cloud providers have similar libraries with which connections can be easily established. Crystal infers the cloud provider from the remote file path. The file path also gives insights into the file owner (user with pre-configured access token) and the blob container that includes the file.

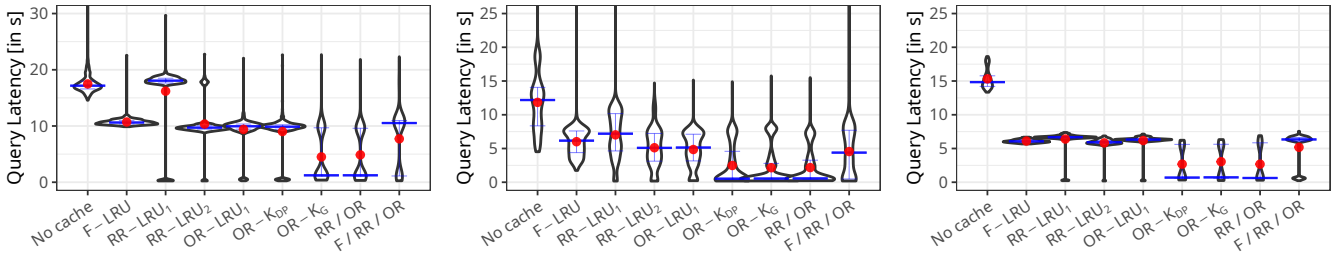
6 EXPERIMENTAL EVALUATION

In this section, we evaluate Crystal as an acceleration unit for Spark, and further report an experiment with Greenplum to show Crystal's generality. Our experiments utilize a single compute node that is connected to standard Azure Blob Storage. The blob storage uses the pre-selected configuration of standard storage and hot tier. All experiments were performed on the DS14_v2 virtual machine. This instance features 16 cores with 112 GB main memory. It comes with 224 GB premium (SSD) storage attached and has a maximum network bandwidth of 12 Gbit/s. The Apache Spark experiments run on version 3.0.1 pre-built for Hadoop 3.2. Our software stack includes Apache Arrow 3.0.0 and azure-storage-cpplite (be490ed).

6.1 Datasets and Caching Strategies

We test workloads comprising real-world data and benchmarks. Following prior work [18], we synthesize queries that contain a mix of range filters and equality filters. Each query is associated with a query type. Within one type, all queries evaluate the same question on a different region of data. For all workloads, we define 5 query types that drill down into distinct combinations of columns.

Lineitem: We generated TPC-H with a scale factor of 50. As lineitem is the main fact table, we use it to schedule predicated



(a) Lineitem (cold cache) (b) Taxi (cold cache) (c) Stocks (after cache warm-up)

Figure 8: Violin plots of the regions workload. The blue bars report the 25th, 50th, and 75th percentiles, the red dot the mean.

Table 1: Datasets Statistics in MB.

Datasets	TPC-H	Lineitem	Taxi	Stocks
Raw	54,344	37,701	32,473	19,100
Parquet	16,822	10,915	6,858	4,021

queries. Query predicates build upon typical predicates of lineitem and answer questions such as how much revenue was created in the year with low-taxed products.

NYC Taxi: The New York City Taxi dataset includes detailed taxi trips between different regions and locations of the city [22]. Each ride is associated with the duration, price, range, start time, end time, and locations. We drill down on this table to analyze multiple aspects of the data. An example query answers how much tip was accumulated for a region of fares during a certain date range.

Historical Stock Prices: Our second real-world dataset contains historic stock prices of the New York Stock Exchange [39]. With information on open, close, volume, and dates many analytics on stock changes are possible. We execute queries that for example help to determine which stocks had the highest intraday changes in the last year while being traded with high volume.

For a fair comparison, the cache size is given in percentage of the full Parquet remote data. The default value is 20%. Table 1 summarizes statistics on the size of the raw files and the converted and snappy encoded Parquet files.

Crystal supports a wide variety of caches and caching policies:

- No Cache uses vanilla Spark without any code changes
- File Cache (F) caches on a block / file level (traditional cache)
- Requested Region Cache (RR) caches semantic regions eagerly
- Oracle Region Cache (OR) caches semantic regions lazily according to decisions of an oracle

Crystal's caching policies include LRU, LRU-k, DP Knapsack (K_{DP}), and our novel Overlap Greedy Knapsack (K_G). Both knapsack strategies leverage our automatic approximative merging augmentation to find region supersets. For the experiments, we keep a history of 128 regions (see Section 6.8). In the result plots, we combine the cache handle with a caching policy. For example, *RR-LRU*$_2$ denotes a competitor that uses only the requested region cache with LRU-2 as caching policy. Because the number of combinations is large, we focus on the individual strategies and show only two combinations. First, an equal combination of *F-LRU*, *RR-LRU*$_2$, and *OR-K*$_G$ cache, that all get $\frac{1}{3}$ of the caching space. Second, we combine the short-term *RR-LRU*$_2$ and the long-term *OR-K*$_G$ with 95% of the cache denoted to OR. *RR / OR* uses mostly the OR cache because RR is only used to cope with changes that are not yet considered by OR (refresh latency). We propose to use *RR / OR* as it provides superior long-term knowledge because of our overlap-aware *OR-K*$_G$ while being able to adapt quickly to short-term changes with *RR-LRU*$_2$.

6.2 Regions of Query Accesses

This scenario features a regional access pattern which can help reduce future request latencies. Each of the query types spans a region that contains 10 - 15% of the tuples. A query reads between 8 - 13% of a random sample of the region (~1% of the tuples).

The regions workload explores a region by individual queries that overlap in some of the dimensions. The overall union of all the queries within a region represents a large fraction of the region's spanning space. We decide on the region before an individual query is chosen. The regions are accessed in a non-uniform pattern as these span a large percentage of the remote data. Next, one of the 200 pre-computed queries per region is picked randomly. The query parameters are chosen uniformly inside the borders of its type region. The region experiments schedule 400 queries in total. We have made the queries available at [12].

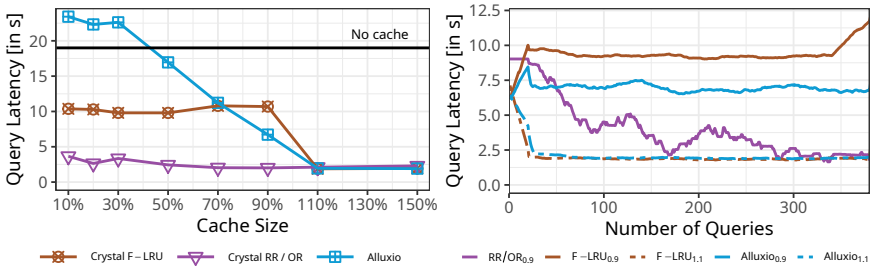
Figure 8 shows this long-term knowledge workload for the lineitem, taxi, and stocks datasets. Note that, the lineitem and taxi experiments run on cold caches. As violin plots give insights into the distribution of the queries, we prefer them over box plots. The shape represents the density of the observations at this value (smoothed by a kernel). Violin plots also encode percentiles and the mean.

Overall, we see significant improvements of the oracle region caches. The greedy knapsack and its *RR/OR* variant with the overlap adaptions outperform the competitors in all three workloads. Because *OR-K*$_G$ benefits most from augmentation, more general regions of interest are found. Especially, at the end of the experiment better caches are used to significantly speed up query processing.

To demonstrate the effectiveness of the better caches, we show the stocks queries after warm-up. At the warm-up phase, the semantic caches (*OR* and *RR-LRU*$_2$) have similar performance. They can capture some of the frequent queries, but fail to generalize. Over time, *OR* learns to cache a better subset. The improvements are shown in Figure 8c. We analyze the cache refresh latency of oracle region caches in Section 6.8.

6.3 Crystal vs. Block Caching

To highlight the performance benefits of Crystal compared to traditional caches, we run the regions workload with different cache sizes. We compare Crystal's *RR / OR* approach with a traditional



(a) Mean lineitem regions performance on varying cache sizes after a warm-up phase.

(b) Lineitem regions mean latency over time with cache sizes 90% (solid) and 110% (dashed).

Figure 9: Comparison against block-based (Alluxio) and file-based caching.

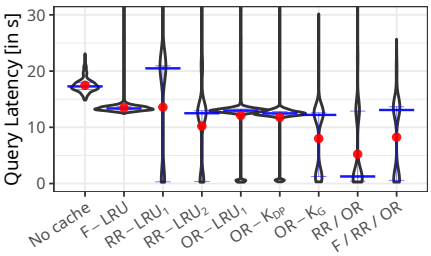


Figure 10: Lineitem regions workload with hot event queries.

file cache (*F-LRU*) and a block-level cache. The block-level cache is represented by Alluxio – which is a widely used accelerator for analytics on disaggregated storage [1, 32, 33].

Figure 9a shows the average lineitem regions performance for different cache values which is denoted by the percentage of the original data. This experiment was conducted after the warm-up phase. Crystal's *RR / OR* is able to learn the set of data needed to accelerate the workload even with limited cache available. Alluxio is initially slower than the vanilla Spark implementation. The larger the cache size, the better Alluxio performs as less blocks have to be evicted. With 110% of the data the maximum performance is reached as all blocks reside on local SSD after the first hit. Only in this extreme scenario, Alluxio is able to match the performance of *RR / OR*. The file based *F-LRU* benefits from an optimized downloading of Parquet files from Azure Blob Storage. Only at very high cache sizes, *F-LRU* is able to improve its performance further.

Even in the high cache size scenarios, *RR / OR* can match the query latencies after warm-up. Figure 9b plots the performance over time compared to the 90% and 110% versions of Alluxio and *F-LRU* with cold caches. Crystal's *RR / OR* is only plotted for 90% (no visible difference to 110%). The traditional caches directly reach the maximum performance whereas *RR / OR* learns the best set of tuples over time. After a sufficiently large number of queries seen, *RR / OR* achieves a similar performance even in the 110% scenario.

6.4 Benefit of RR/OR

In the previous experiments, long-term knowledge was sufficient to solve the scenarios optimally. Real-world workloads often incur a combination of short-term bursts of queries and long-term trends. Similar to a news-breaking event, we reformulate our regions experiment. Regions are still accessed with their respective distribution but additional queries arrive that target different data. These queries access a specific event which first spikes and then falls back to normal. In our scenario, event queries occur up to 8 times (out of 200 queries), before they disappear and another event starts. We schedule $\frac{2}{3}$ regions query and $\frac{1}{3}$ event queries.

The results are shown in Figure 10. Long-term caches (*OR*) fail to catch the events due to their latency constraints. Without long-term knowledge, only event queries can be handled. Although *RR-LRU*$_1$ performs better on short-term queries, the long-term deficits of *RR* caches show that only combined strategies with *RR* and *OR* can overcome both issues. *RR/OR* uses a small short-term cache that catches spikes in reoccurring queries and retains the same performance for long-term trends as *OR* caches.

6.5 Changing Regions

Besides the hot region detection, it is important that caches are refreshed if the workload changes. This experiment takes the regions workload and applies a region change after 400 queries (offset to random value of zipfian) and favors the current region ($\theta = 2$).

Figure 11 depicts the changing regions workload for the lineitem and taxi data. For a better visibility, we only show the semantic base algorithms. The performance is shown as moving average ($n = 40$ queries) over the number of queries. After 400 queries, the hotspot region changes. Both workloads are dominated by *OR-K*$_G$ which is able to reduce runtime while being able to quickly adapt. On the other hand, *OR-K*$_{DP}$ has a long latency until it reaches its optimum again which is easily visible in the lineitem plot. *OR-LRU*$_1$ could adapt fast, however, the overall performance gains are smaller than for *OR-K*$_G$. The same is true for short-term caches such as *RR-LRU* because they fail to generalize to the semantic regions of interest.

6.6 Crystal's Database Agnostic Design

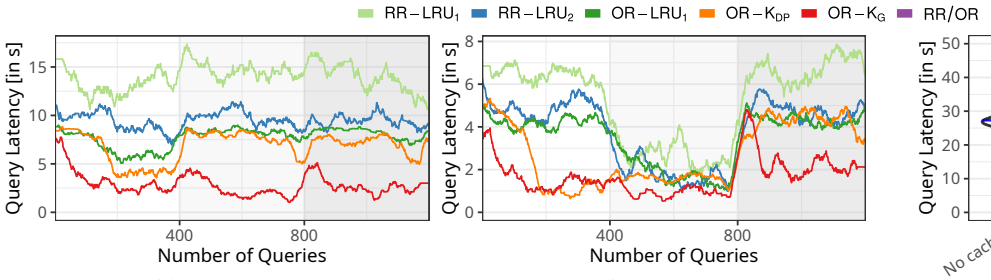
To evaluate our design principle of generality, we show that Crystal can be used as a storage layer for a different DBMS, Greenplum (6.16) [51]. The details of the connector are discussed in Section 5.

Figure 12 depicts the lineitem region workload executed within the Greenplum database. For optimal performance, we configured Greenplum such that it uses as many PostgreSQL workers (segments) as the compute node features hardware threads. Although the performance of Greenplum is slightly worse compared to Spark, the relative benefits of using Crystal stay similar.

6.7 TPC-H

Figure 13a shows the cumulative distribution function for queries generated with the TPC-H Q6 template. We perform the selection according to TPC-H but change the distribution of the shipdate. In many real-world workloads, accesses on date dimensions are heavily skewed. We mimic this behavior by sampling the shipdate from the zipfian distribution. The experiment validates that the *(RR/) OR-K*$_G$ caches capture the hot regions.

Further, we test all 22 queries of TPC-H which are executed twice in this experiment. Due to the construction of TPC-H, only a few queries have atomic predicates on the large tables. Most restrictions come from join conditions which currently cannot be pushed down. Thus, many queries need to get the full table from the remote. Nevertheless, Figure 13b shows that the *OR* approaches are able to improve the mean performance by up to 20%.



(a) Lineitem (b) Taxi

Figure 11: In each third the hot region is changed, i.e. the zipfian is moved to the next region. The plots show the moving average with window size 40 queries.

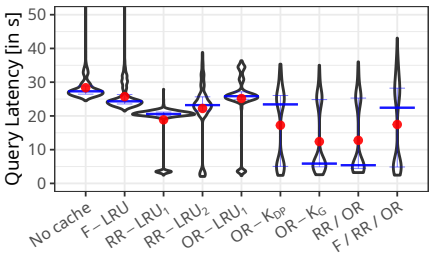
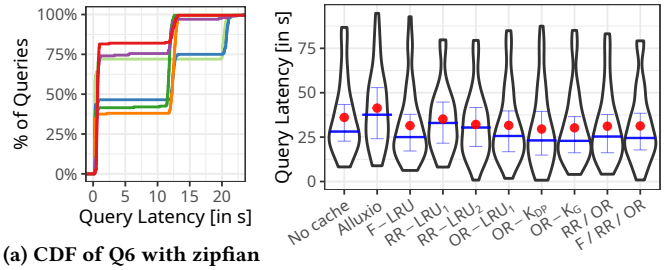
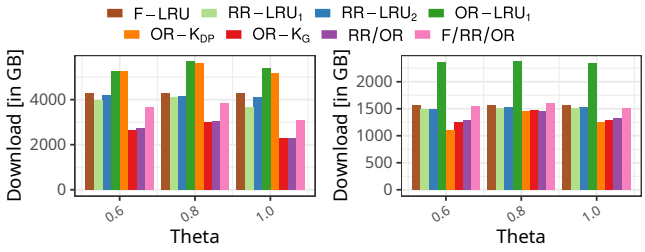


Figure 12: Lineitem regions workload benchmarked on Greenplum.



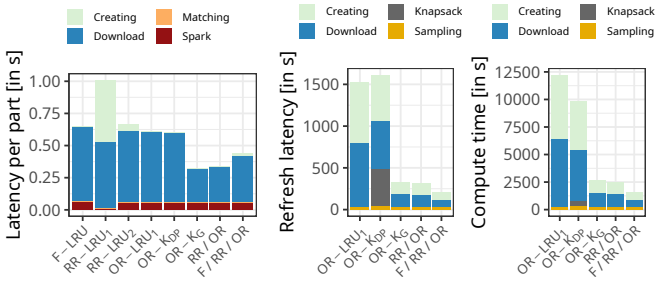
(a) CDF of Q6 with zipfian distributed date. (b) Violin plots of all 22 queries.

Figure 13: Performance on TPC-H.



(a) Regions - Lineitem (b) Regions - Stocks

Figure 15: The total network traffic used by each strategy plotted on a varying theta.



(a) Query latency per Parquet part (b) Oracle: cache refresh latency (c) Oracle: sum of compute time

Figure 14: Operation break-down on the online end-to-end and the offline background tasks for lineitem regions.

6.8 Microbenchmarks

To get more insights into the behavior of the caches, we conducted several microbenchmarks. This section evaluates the breakdown of operations, the different parameters, and the network footprint.

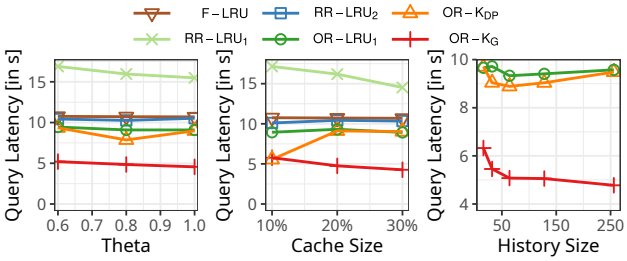
Figure 14 shows the breakdown of the individual operations (mean) during query processing. We split the plots into online (Figure 14a) and offline threads (Figures 14b and 14c). The online thread receives the region request and responses with either cached regions or the raw files downloaded into ramdisk. The largest runtime contributors are the downloading of files and the eager computation. Note that, Figure 14a does not rely on the DBMS to handle the eager file creation. The plot shows that matching does not introduce latency overhead. The background threads are only relevant for oracle region caches. Crystal spawns a low-priority daemon that periodically recomputes the cache. Figure 14b shows the latency until a new cache is computed, Figure 14c sums up the wall-clock of all computing threads. As the threads run at a high nice level, the scheduler does not often consider these threads as soon as load hits the compute node. This results in higher latencies, however, the query processing of the DBMS is not negatively affected by Crystal's background task. The *OR-K_{DP}* has an additional high latency due to the computation of the knapsack. On the other hand, *(RR/) OR-K_G* has low computation effort while caching useful regions. Thus, *(RR/) OR-K_G* downloads and computes less unnecessary files which results in superior background time. *OR-LRU_1* often downloads infrequent regions that will be discarded in the next iteration. Hence, additional and unnecessary computation is introduced.

Figure 15 shows the network footprint for different θ values. *RR/OR-K_G* reduces the network traffic in comparison to our file cache baseline. As more frequent regions are accessed, less traffic is needed. *OR-LRU_1* is prune to downloads of infrequent regions.

Lastly, we vary the used parameters throughout the experimental evaluation in Figure 16. For the regions workload, we show that the algorithms perform similarly for different skew θ of the regions. Nevertheless, the oracle region strategies outperform the baseline in all scenarios. The cache size experiment shows that the oracle algorithms pick the most valuable regions first. Even with a cache that uses only 10% of the data size, good performance is achieved. Smaller history sizes (< 64) decrease the performance as too few queries are collected for predicting the workload. Large history sizes are slower in adopting to new frequent regions.

7 RELATED WORK

The basic idea behind Crystal is to cache and reuse computations across multiple queries. This idea has been explored in a large body of research work including at least four broad lines of research: materialized view, semantic caching, intermediate results reusing, and



(a) Theta. (b) Cache Size. (c) History Size.

Figure 16: Microbenchmarks on lineitem regions.

mid-tier database caching. In general, Crystal differs from previous work in some or all of the following ways: 1) integrating Crystal with a DBMS requires no modification to the DBMS; 2) Crystal focuses on caching views at the storage layer, and can be used across multiple DBMSs; 3) Crystal can automatically choose cached views based on a replacement policy, which takes into account the semantic dependencies among queries. Below, we discuss the key differences between Crystal and previous work in each line of the four aforementioned research areas.

Materialized View. Materialized view is a well-known technique that caches the results of a query as a separate table [20, 44, 45]. However, unlike Crystal, views that need to be cached or materialized are often defined manually by users (e.g., a DBA). Additionally, implementing materialized views in a DBMS is a time-consuming process, requiring advanced algorithms in the query optimizer to decide: 1) if a given query can be evaluated with a materialized view; and 2) if a materialized view needs to be updated when the base table is changed.

Semantic Caching. Semantic caching was first proposed in Postgres [47], and was later extended and improved by a large body of work [11, 14, 17, 29, 30, 42, 43]. This technique also aims to cache the results of queries to accelerate repeated queries. Similarly to Crystal, a semantic cache can automatically decide which views to keep in the cache, within a size budget. This decision is often made based on a cost-based policy that takes several properties of views into consideration such as size, access frequency, materialization cost. However, this approach caches the end results of entire queries, while Crystal caches only the intermediate results of the selection and projection operators of queries. The cached view of an entire query is especially beneficial for repeated queries, but on the other hand decreases the reusability of the cached view, i.e., the chance that this view can be reused by future queries. While most work in this area does not take into account overlap of cached views, some work [14, 17] does explore this opportunity. Dar et al. proposed to split overlapping queries into non-overlapping regions, and thus enable semantic cache to use traditional replacement policies to manage the (non-overlapping) regions [14]. However, this approach could result in a large number of small views, incurring significant overhead to process as we showed in Sec 3.2. Maintaining non-overlapping views is also expensive, as access to an overlapping view may lead to splitting the view and rewriting the cached files. Chunk-based semantic caching [17] was proposed to solve this problem, by chunking the hyper space into a large number of regions that are independent to queries. However, the chunking is pre-defined and thus is static with respect to the query patterns.

Intermediate Results Reusing. Many techniques have also been developed to explore the idea of reusing intermediate results rather than end results of queries. Some of these techniques [49, 50] share the intermediate results across concurrent queries only, and thus impose limitations on the temporal locality of overlapping queries. Other work [19, 25–27, 38, 41] allows intermediate results to be stored so that they can be reused by subsequent queries. Similarly to Crystal, these techniques also use a replacement policy to evict intermediate results when the size limit is reached. However, these techniques require extensive effort to be integrated with a DBMS, whereas integrating Crystal requires only a lightweight database-specific connector. Additionally, a Crystal cache can be used with and share data across multiple DBMSs.

Mid-tier Database Caching. Another area where views can be cached and reused is in the context of multi-tier database architecture, where mid-tier caches [2, 10, 31] are often deployed at the mid-tier application servers to reduce the workload for the backend database servers. As mid-tier caches are not co-located with DBMSs, they usually include a shadow database at the mid-tier servers that mirrors the backend database but without actual content, and rely on materialized views in the shadow database to cache the results of queries. Unlike Crystal, the definition of the cached views in a mid-tier cache needs to be pre-defined manually by users, and it is difficult to change the cached views adaptively.

Finally, many vendors have developed cache solutions for big data systems to keep hot data in fast local storage (e.g., SSDs). Examples include the Databricks Delta Cache [9, 15], the Alluxio [1] analytics accelerator, and the Snowflake Cache Layer [13]. These solutions are based on standard techniques that simply cache files at the page or block level and employ standard replacement policies such as LRU. Compared to these standard approaches, Crystal is also a generic cache layer that can be easily integrated with unmodified big data systems, but has the flexibility to cache data in a more efficient layout (i.e., re-organizing rows based on queries) and format (i.e., Parquet), which speeds up subsequent query processing.

8 CONCLUSION

Cloud analytical databases employ a disaggregated storage model, where the elastic compute layer accesses data on remote cloud storage in columnar formats. Smart caching is important due to the high latency and low bandwidth to remote storage and the limited size of fast local storage. Crystal is a smart cache storage system that co-locates with compute and can be used by any unmodified database via data source connector clients. Crystal operates over semantic data regions, and continuously adapts what is cached locally for maximum benefit. Results show that Crystal can significantly improve query latencies on unmodified Spark and Greenplum, while also saving on bandwidth from remote storage.

ACKNOWLEDGMENTS

Dominik Durner has received funding in part from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286). We would also like to thank the anonymous reviewers and Donald Kossmann for their feedback and support of this work.

REFERENCES

[1] Alluxio. 2021. Alluxio - Data Orchestration for the Cloud. https://www.alluxio.io/. accessed: 2021-07-16.

[2] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. 2003. Cache Tables: Paving the Way for an Adaptive Database Cache. In *VLDB*. 718–729.

[3] Amazon. 2020. Amazon S3 Cloud Storage. https://aws.amazon.com/s3/. accessed: 2021-07-16.

[4] Amazon. 2020. Amazon S3 Select. https://aws.amazon.com/blogs/aws/s3-glacier-select/. accessed: 2021-07-16.

[5] Apache. 2021. Apache Arrow. https://arrow.apache.org/. accessed: 2021-02-17.

[6] Apache. 2021. Apache Avro. https://avro.apache.org/. accessed: 2021-02-17.

[7] Apache. 2021. Apache Parquet. https://parquet.apache.org/. accessed: 2021-02-17.

[8] Apache. 2021. Gandiva for Apache Arrow. https://arrow.apache.org/blog/2018/12/05/gandiva-donation/. accessed: 2021-02-17.

[9] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *PVLDB* 13, 12 (2020), 3411–3424.

[10] Christof Bornhövd, Mehmet Altinel, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. 2004. Adaptive Database Caching with DBCache. *IEEE Data Engineering Bulletin* 27, 2 (2004), 11–18.

[11] Chung-Min Chen and Nick Roussopoulos. 1994. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *EDBT*. 323–336.

[12] Crystal. 2021. Regions workload queries. https://aka.ms/crystal-queries. accessed: 2021-07-16.

[13] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. 215–226.

[14] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In *VLDB*. 330–341.

[15] Databricks. 2021. Delta Cache. https://docs.databricks.com/delta/optimizations/delta-cache.html. accessed: 2021-02-17.

[16] Databricks. 2021. Introduction to Data Lakes. https://databricks.com/discover/data-lakes/introduction. accessed: 2021-02-17.

[17] Prasad Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. 1998. Caching Multidimensional Queries Using Chunks. In *SIGMOD*. 259–270.

[18] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *PVLDB* 14, 2 (2020), 74–86.

[19] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, and Tim Kraska. 2017. Revisiting Reuse in Main Memory Database Systems. In *SIGMOD*. 1275–1289.

[20] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing Queries Using Materialized Views: A practical, scalable solution. In *SIGMOD*. 331–342.

[21] Goetz Graefe. 2009. Fast loads and fast queries. In *DaWaK*. 111–124.

[22] Evan Hallmark. 2019. Daily Historical Stock Prices. https://www.kaggle.com/ehallmar/daily-historical-stock-prices-1970-2018. accessed: 2021-01-17.

[23] Karl Holub. 2021. The Overlapped Hyperrectangle Problem. https://kholub.com/projects/overlapped_hyperrectangles.html. accessed: 2021-02-17.

[24] Oscar H Ibarra and Chul E Kim. 1975. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM (JACM)* 22, 4 (1975), 463–468.

[25] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. 2009. An architecture for recycling intermediates in a column-store. In *SIGMOD*. 309–320.

[26] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. *PVLDB* 11, 7 (2018), 800–812.

[27] 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 *SIGMOD*. 191–203.

[28] Srikanth Kandula, Laurel Orr, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates through Joins in Big-Data Clusters. *PVLDB* 13, 3 (2019), 252–265.

[29] Donald Kossmann, Michael J. Franklin, and Gerhard Drasch. 2000. Cache investment: integrating query optimization and distributed data placement. *TODS* 25, 4 (2000), 517–558.

[30] Yannis Kotidis and Nick Roussopoulos. 1999. DynaMat: A Dynamic View Management System for Data Warehouses. In *SIGMOD*. 371–382.

[31] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. 2004. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *ICDE*. 177–188.

[32] Haoyuan Li. 2018. *Alluxio: A virtual distributed file system.* Ph.D. Dissertation. UC Berkeley.

[33] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *SoCC*. 6:1–6:15.

[34] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *SIGMOD*. 2530–2542.

[35] Microsoft. 2020. Azure Data Lake Storage Query Acceleration. https://docs.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-query-acceleration. accessed: 2021-07-16.

[36] Microsoft. 2020. Azure Storage - Secure cloud storage. https://azure.microsoft.com/en-us/services/storage/. accessed: 2021-07-16.

[37] Microsoft. 2021. Azure Storage C++ Client Library (Lite). https://github.com/Azure/azure-storage-cpplite. accessed: 2021-07-16.

[38] Fabian Nagel, Peter A. Boncz, and Stratis Viglas. 2013. Recycling in pipelined query evaluation. In *ICDE*. 338–349.

[39] City of New York. 2021. New York City TLC Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page. accessed: 2021-03-01.

[40] Oracle. 2021. Database Data Warehousing Guide: Using Zone Maps. https://docs.oracle.com/database/121/DWHSG/zone_maps.htm. accessed: 2021-07-16.

[41] Luis Leopoldo Perez and Christopher M. Jermaine. 2014. History-aware query optimization with materialized intermediate views. In *ICDE*. 520–531.

[42] Peter Scheuermann, Junho Shim, and Radek Vingralek. 1996. WATCHMAN : A Data Warehouse Intelligent Cache Manager. In *VLDB*. 51–62.

[43] Junho Shim, Peter Scheuermann, and Radek Vingralek. 1999. Dynamic Caching of Query Results for Decision Support Systems. In *SSDBM*. 254–263.

[44] Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. 1998. Materialized View Selection for Multidimensional Datasets. In *VLDB*. 488–499.

[45] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. 1996. Answering Queries with Aggregation Using Views. In *VLDB*. 318–329.

[46] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *VLDB*. 553–564.

[47] Michael Stonebraker, Anant Jhingran, Jeffrey Goh, and Spyros Potamianos. 1990. On Rules, Procedures, Caching and Views in Data Base Systems. In *SIGMOD*. 281–290.

[48] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*. 1115–1126.

[49] Kian-Lee Tan, Shen-Tat Goh, and Beng Chin Ooi. 2001. Cache-on-Demand: Recycling with Certainty. In *ICDE*. 633–640.

[50] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. 2019. Intermittent Query Processing. *PVLDB* 12, 11 (2019), 1427–1441.

[51] VMware. 2021. Greenplum Database - Massively Parallel Postgres for Analytics. https://www.greenplum.org/. accessed: 2021-07-16.

[52] VMware. 2021. Greenplum Platform Extension Framework (PXF). https://docs.greenplum.org/6-8/pxf/overview_pxf.html. accessed: 2021-07-16.