Cool: a COhort OnLine analytical processing system

Zhongle Xie[†], Hongbin Ying[§], Cong Yue[†], Meihui Zhang[‡], Gang Chen[⋆], Beng Chin Ooi[†],

[†] {*zhongle, yuecong, ooibc*}*@comp.nus.edu.sg, National University of Singapore*
[§] *yinghongbin@mzhtechnologies.com, MZH Tech*
[‡] *meihui_zhang@bit.edu.cn, Beijing Institute of Technology*
[⋆] *cg@zju.edu.cn, Zhejiang University*

***Abstract*—With a huge volume and variety of data accumulated over the years, OnLine Analytical Processing (OLAP) systems are facing challenges in query efficiency. Furthermore, the design of OLAP systems cannot serve modern applications well due to their inefficiency in processing complex queries such as cohort queries with low query latency. In this paper, we present *Cool*, a cohort online analytical processing system. As an integrated system with the support of several newly proposed operators on top of a sophisticated storage layer, it processes both cohort queries and conventional OLAP queries with superb performance. Its distributed design contains minimal load balancing and fault tolerance support and is scalable. Our evaluation results show that *Cool* outperforms two state-of-the-art systems, MonetDB and Druid, by a wide margin in single-node setting. The multi-node version of *Cool* can also beat the distributed Druid, as well as SparkSQL, by one order of magnitude in terms of query latency.**

***Index Terms*—Cohort Analysis, OLAP, Distributed System**

I. Introduction

A wide variety of data is accumulated by companies and organizations during their operation. For example, e-commerce websites collect data about sales products and customer purchasing histories while Health Information Systems (HIS) keep electronic medical records such as lab-test results and admission histories on patients. To fathom such data and extract deep insights, OnLine Analytical Processing (OLAP) system is often used by analysts. In practice, the analytical system treats the data that has been imported through "Extract, Transform, Load" (ETL) processes, as conceptual "data cube", and performs cube queries to support drill-down and roll-up analysis [1]–[3].

A cohort analysis is gaining popularity as a method of analyzing a metric by comparing the behavior between different groups of users (buyers, patients etc.) [4], [5], and the grouping is done either based on events or the time users start a service. That is, a cohort query is to explore the user behavioral pattern from two major factors, i.e., aging and social changes. Intuitively, the execution of a cohort query is divided into three steps: (1) Find birth user cohort, which is to locate users experiencing similar given events or characteristics and group them into corresponding cohorts. (2) Calculate the ages of the users in the cohorts, which is to, for each user in the cohort, cut the corresponding records into diverse segments by given delimiters along time axis. (3) Aggregate the metrics, which is to measure the value on each segment by a given aggregator.

TABLE I: Glucose Cohort vs. Readmission

#Patients / Age / Cohort	Month after discharge					
	1	2	3	4	5	6
$<$ 99 mg/dL (130)	10	4	3	5	1	1
100-130 mg/dL (350)	30	17	9	6	3	1
130-160 mg/dL (90)	22	20	15	14	11	5
160-190 mg/dL (50)	15	13	9	8	2	1
$>$ 191 mg/dL (12)	12	-	-	-	-	-

Take the following scenario in healthcare industry as an instance [6]: *"A doctor is keen on discovering the relationship between diabetic patients' readmission and their blood glucose when the patients are admitted and discharged by the hospital"*. The query can be processed by grouping patients into cohorts based on their blood glucose lab-test value and comparing the cohorts by the number of readmissions along different time intervals. An illustrative result is shown in Table I, where the total number of patients of each cohort is recorded in parentheses. The doctor can easily find interesting phenomenons from the table, such as the fact that the higher glucose level a patient has, the more likely the patient is readmitted by the hospital.

Cohort query has also been used in retention analysis of web applications [7]–[9]. However, traditional OLAP systems have not been designed to support such queries. As reported in [6], the query latency of a typical cohort query executed in *MonetDB* is one order of magnitude slower than performed in a state-of-the-art specialized engine called Cohana, albeit its inability of running cube queries. As a result, it is often difficult and slow to run such queries and cube queries simultaneously on existing OLAP systems. For example, running a drill-down analysis while checking the customer retention with respect to the customer occupations may result in a long wait just because of inefficient processing. Such composite queries that integrate OLAP queries and cohort queries together are frequently applied in real-world applications, such as funnel analysis in Mixpanel [10].

In addition, the rapid and continuous increase of data

volume and variety over the years have caused a sharp increase in both storage space requirement and query latency [11]. For emerging patabyte scale OLAP systems [11], [12], the huge size of the cube causes frequent disk accesses and hence significantly hurts their performance.

Most existing solutions tackle the above challenges separately, using independent designs and layouts in system implementation. In particular, they either make use of columnar architecture to scale up [13] or take advantages of distributed processing for specific queries [14], [15]. Realistically, it is non-trivial integrating them as a general system to leverage the advances of different solutions. For instance, to process a successive query whose input is the output of another precedent query, redundant storage consumption and transformation cost can be easily incurred once the processors for the two queries have incompatible input/output format and resort to different storage layouts.

In this paper, we propose *Cool*, a cohort OLAP system. *Cool* supports both conventional OLAP queries and emerging cohort queries as an integrated platform for data analytics. Specifically, the following contributions are made:

- We provide an integrated solution to support both conventional OLAP query and cohort query. The queries are formatted with a pre-defined and consolidated JSON format. With the support of two database operators for OLAP query, i.e., *metaChunk* selector and *dataChunk* selector, and three operators for cohort query, including birth selector, age selector and cohort aggregator, the system is able to perform both types of queries with similar processing flows.
- We also present a sophisticated storage layout for the system with optimization in query processing and space consumption. With configurable and compressed pre-computation results embedded in storage, *Cool* can trade little space for large execution boost when processing OLAP queries.
- We scale the system with an efficient distributed processing architecture. With the help of HDFS and Zookeeper, the system is able to effectively balance the workload and recover from the failures.
- We experimentally compare *Cool* with two state-of-the-art analytical systems, namely *Druid* and *MonetDB*, in single-node settings. The results show that *Cool* not only offers more flexibility than the two baselines, but also performs the best in terms of query efficiency and space consumption. Our evaluation on distributed environment also indicates that *Cool* can be one order of magnitude faster than two state-of-the-art systems, namely *Spark-SQL* and distributed *Druid*.

The rest of the paper is organized as follows. Section II presents the state-of-the-art work in the literature on OLAP systems. The single-node architecture of *Cool* is introduced in Section III while the distributed version is proposed in Section IV. We evaluate *Cool* with our baselines in both single-node and distributed environments in Section V and conclude this work in Section VI.

II. Related Work

Due to the fast growing data volume, many OLAP systems have been designed and proposed to process analytical queries efficiently. Existing systems can be broadly divided into two categories, i.e., conventional OLAP systems built on top of database systems and the emerging query engines with optimization on specific query types.

Conventional OLAP Systems. Due to the poor performance of row-oriented stores on analytical workloads, columnar store is proposed to improve the speed of the query processing in the literature. Since OLAP systems ought to scan a large volume of records for a limited set of columns, columnar store benefits by avoiding loading data of irrelevant columns that do not contribute to the final results. Moreover, the size of the scanned data can be further shrunk by compression schemes such as dictionary encoding [16], run length encoding (RLE) [1], row re-ordering [17], etc. Representative systems include MonetDB [13], [18], MariaDB [19] and HBase [20].

However, columnar stores have shortages that the performance of data insertion and point query are not as good as traditional row-oriented databases. Consequently, proposals have been made to combine the key ideas of row-oriented store and columnar store within one system [21]. The examples include MemSQL [22], Oracle Database In-memory [23], Hyper [24] and SAP HANA [25]. Such systems can maintain data tables in either column-oriented or row-oriented format. A hybrid storage plan, organizing the data in both row-oriented and column oriented manners, is also employed.

To handle enormous amount of data, several offline OLAP approaches have also been proposed in the literature. Distributed processing systems, such as Hive [26], Qylin [27] and Impala [14], trade off between computing resources and overall processing performance. In practice, their execution model relies on the execution efficiency of worker nodes. Consequently, load balancing and network communication are two important optimization objectives [28], and Google's Dremel is one such example [29]. Notwithstanding, loading the data into the query processing engine often results in long set up time before a query task can run [30].

Emerging Query Engines. There are increasing number of emerging classes of queries engendered by new applications, data, and business models [31] that cannot be efficiently supported by conventional OLAP systems. Consequently, new systems are designed to meet these demands. For example, Pinot [32], a query processing engine used in Linkedin, offers near real-time response for iceberg queries [33] with respect to the latest data injected into the OLAP system while Druid [15], an open source data store which shares similar ideas, supports flexible filters and efficient aggregations of the injected records with time-dimension optimization. Data cube [34] is widely used for queries involving multiple dimensions from the dataset. To tackle the challenge of expensive query processing, an index called star-tree is used to support efficient OLAP queries irrespective of data distribution [35].

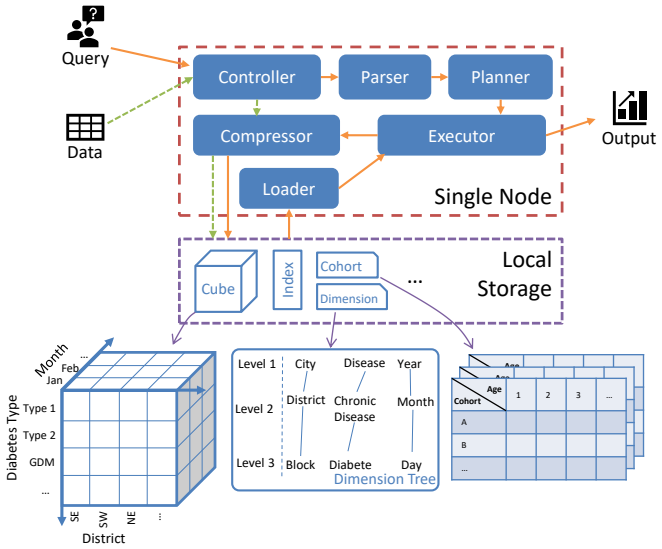


Fig. 1: *Cool* Single-node System Architecture

To improve the performance of such systems, diverse compression algorithms have been employed. PowerDrill employs a two-level dictionary compression scheme to enable processing of trillion record cells in seconds [16] while condensed cube [36] reduces the size of data cubes to improve the overall computation time. Furthermore, with the increasing demand for real-time OLAP queries, it is also important to improve data injection efficiency. Spark has been extended to answer stream queries [37] and Shark further improves its performance by leveraging distributed memories [38].

The first specialized engine for cohort queries, called Cohana [6], [39], reports an extraordinary performance when compared to the implementation using traditional SQL solutions. Yet, the cohort query formalized in this work only supports a single birth event definition and time-dimension age delimiters, which significantly constrains the diversity of the cohorts and therefore affects its usability [40]. Compared to Cohana, *Cool* differs in three aspects: (1) *Cool* uses a more general definition for cohort queries. It employs a sequence of birth events to find the valid users and implements both temporal age delimiter and event-based age delimiter, which will be discussed in Section III-A. (2) *Cool*, as an integrated platform, also supports conventional cube queries efficiently while Cohana does not. (3) For big data scenarios, *Cool* is able to leverage the computing power of multiple nodes while Cohana depends on standalone servers.

III. Single-node Architecture

We first present the single-node architecture of *Cool* in this section, followed by the scalable distributed processing architecture in the next section. There are six components in the system as illustrated in Figure 1, consisting of loader, controller, parser, planner, compressor and executor.

A. Data Model

Similar to conventional databases, *Cool* organizes data records in tables. Each table is maintained by a user-defined schema, consisting of multiple columns called fields or dimensions. A dimension file recording the hierarchy of several dimensions is also given by the users in order to support cube queries. Each dimension of the table is bound to a field type describing the format of the values. The primitive types include varied-length integer, float, string, boolean, time and event. The first four are similar to underlying database types while the remaining types are specifically defined by *Cool*. Time deals with the timestamps of the records and event is a particular string representing user actions or behaviors. A dimension tree is constructed in system setup guided by the dimension file and persisted in storage, as shown in Figure 1. *Cool* employs a hybrid orientation plan for data storage, as depicted in Figure 2. The tables are horizontally split into different partitions called *cublets*. Each cublet consists of multiple chunks, where the *metaChunk* contains all the values for a corresponding field in this cublet.

In *Cool*, all the queries are written in JSON format with a pre-defined syntax. Two types of queries are supported: (1) **OLAP Query.** *Cool* can be treated as a conventional OLAP system dealing with cube queries. The system supports basic cube operations, including roll-up, drill-down, pivot and slice and dice, upon data cubes built atop the fields and the dimension tree. *Cool* can also provide responses for iceberg query [33], a prevailing type of query selecting a small number of records which satisfy some given conditions. (2) **Cohort Query.** *Cool* can support an enhanced version of cohort queries. Traditional cohort query, as defined in Cohana [6], can only support aggregations on cohorts born with a single event along a fixed time window. However, *Cool* supports aggregations on cohorts born with a series of events, namely an event sequence, along either a fixed time window or an elastic time window delimited by given events.

Cool supports composite query processing by running a successive query atop the result of a precedent query. The type of the precedent query and the successive query can be any combination of the two aforementioned query types. For instance, a cohort generated in a precedent cohort query can act as a data source for either a successive OLAP query or a successive cohort query. Such composite query is sometimes called *funnel analysis* [10] in the literature. In the implementation, based on the user-defined schema and the dimension tree, data cube and cohort are stored conceptually as the intermediate structures to support such query, as shown in the bottom part of Figure 1. The meaning of "conceptually" here is that *Cool* materializes the matched records of the precedent query as well as the corresponding dimension structure in the storage layer and aggregates the corresponding results in runtime, instead of storing the resultant cohort or data cube tuples directly. By default, such intermediate structures exist in a fixed period and the system can be configured to persist them permanently.

B. Storage Layout

The hierarchy of the cublet is illustrated in Figure 2. A cublet consists of a *metaChunk* and multiple *dataChunk*s. *Cool*

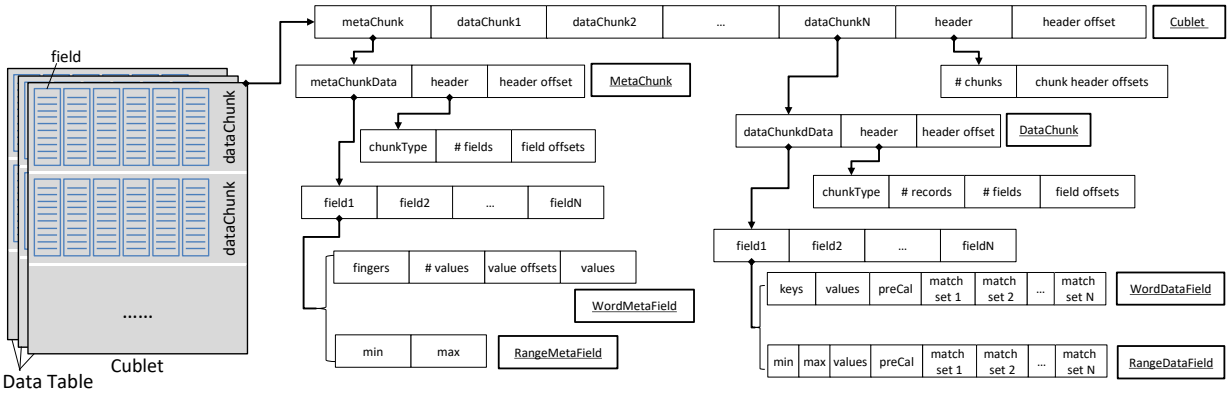


Fig. 2: Storage Hierarchy

transcripts the columnar data into field and stores multiple fields into *dataChunk* with the meta types derived from the user-defined schema. The *metaChunk* is used to store the meta data for each *dataChunk* within the cublet, including the number of contained fields and the range or the encoding structures of the corresponding field, as shown in the figure. The number of *dataChunk*s in the cublet is recorded in the header. The "offset" in the figure records the positions of corresponding headers or fields.

There are two basic meta types for field stores, namely *range* and *word*. The basic types, including integer, float and time, are converted to range field in the storage layer. The remaining types, such as string and event, are treated as word field in *Cool*. The system has different storage plans for the two field stores as shown in the last level of *dataChunk* in the figure, whose details are discussed in the next paragraph. It is worth noting that several match sets are piggybacked when the *dataChunk* is first stored in order to accelerate cube query processing. The number of match sets is configurable in the system setup and such match sets are in fact bitsets indicating whether the record related to the bit equals to the given value.

Cool employs various compression algorithms and encoding schemes based on the storage plan for different field types in order to reduce space consumption. The storage plan for range field is to only keep maximum and minimum values as depicted in Figure 2. In practice, a delta encoding scheme is used on the field to compress the data in such kind of field. Each record is shortened as a delta value against the minimum value to store. Meanwhile, the storage plan for word field is to apply a double-dictionary scheme similar to [16] in order to save the space. Each distinct string of the field is put into a global dictionary and can be indexed by a unique number. Therefore, the range set of the field is converted into a series of numbers. In this way, each chunk can only store the numbers existed in the chunk, namely the local value set. To further reduce space cost, numbers in the local value set are further translated into a local dictionary with shortened numbers, i.e., the fingers in Figure 2, to achieve a higher compression ratio. Besides the two encoding schemes, appropriate compression algorithms are also employed to reduce storage size. For example, bit packing and vectorization are introduced to deal with delta values and dictionary indexing numbers.

Indexes are used in *Cool* to support efficient query processing on the hybrid data orientation design. For example, the prefix tree is employed due to its competitive lookup performance and superb space-saving ability in the aforementioned double-dictionary scheme. Besides tree indexing structures, other auxiliary indexes, including bitsets and hash tables, are efficiently used to facilitate operations with filters, such as slice and dice in OLAP and birth selection in cohort query. Another usage of bitset is the match set piggybacked at the end of the fields as shown in Figure 2. To reduce the consumption of space, we further apply bit packing scheme, namely an RLE compression, upon the match set.

C. Query Processing

In this section, we first introduce how *Cool* utilizes different components shown in Figure 1 to generate same execution paths for both conventional OLAP query and cohort query. Then we briefly describe the predicates and the native operators implemented in *Cool*. Two different processing flows and a summary of the advantages on the system is lastly presented.

1) Execution Path: There are two possible execution paths for the system as shown in Figure 1. The green dashed line indicates the data injection path. The data is imported into the system through an ETL process. After the ETL process, the records for each user are ordered chronologically and fed to a controller. The controller passes the data to a compressor, which applies the aforementioned compression schemes on the data according to the user-defined schema. The compressor stores the compressed data locally for further querying and processing. It should be noted that we set two status, namely "readable" and "unreadable", for the cublets in the system to guarantee eventual consistency of the query output. The new cublets under injection process is set unreadable by the controller, forbidding any query processing, and is set to be readable by the controller when the entire injection task is done.

In Figure 1, the orange solid line depicts the execution path on queries within *Cool*. The controller receives queries in JSON from analysts and passes them to a parser. The parser then transforms the queries into different operators with a

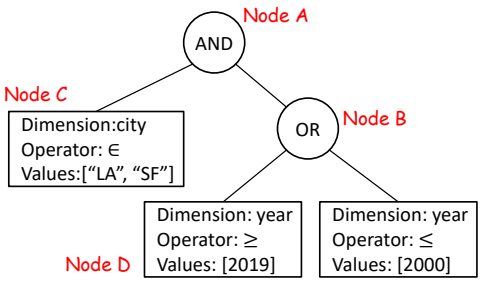


Fig. 3: An example of the predicate tree

system-defined schema. The operators are fed to a planner, generating an execution plan and passing it to an executor. The executor loads the compressed data from the local storage and runs the operators according to the received plan. The result of the query is sent to the user directly. The system can also store the query result for subsequent reference, as shown in the figure where the compressor can be called after the executor finishes its job.

2) Predicates: Predicates are widely utilized to filter inappropriate or redundant records in query processing. There are three types of predicates supported in *Cool*, including *Single*, *Or* and *And*. *Single* is the simplest predicate without any logical operands. *Or* is a joint predicate whose outermost level of logical operand is $\vee$(*or*) while *And* is a joint predicate whose outermost level of logical operand is $\wedge$(*and*).

A predicate tree is generated by the planner shown in Figure 1 and used heavily in query execution. An example of the predicate tree is illustrated in Figure 3, where Node A and Node B are two joint predicate nodes and Node C and Node D are two single predicate nodes. In practice, the node structure of the predicate tree contains following elements:

- **Type.** A string indicating which type of the predicate node is, denoted as t.
- **Operator.** The operator in the predicate, including $\in$, $\notin$, $\leq$, $\geq$, etc. We use symbol α to represent the operator and set it as $null$ for joint predicates, i.e., *Or* and *And*.
- **Dimension.** A string indicating which dimension the predicate should be applied or validated, denoted as d and set as $null$ for joint predicates.
- **Value.** An array contains all the values that need to be checked when applying the predicate, denoted as $\mathbb{V}$ and set to $null$ for joint predicates.
- **Children.** A pointer array contains the operands of joint predicates, denoted as $\mathbb{C}$ and set to $null$ for single predicates.

3) Key Operators: Several operators are implemented for query processing in *Cool*. For OLAP queries, *metaChunk* selector and *dataChunk* selector, are implemented.

metaChunk Selector $\sigma^M_{\mathcal{P}}$. The *metaChunk* selector is used during the scanning of *metaChunk* in query processing. The input of the operator consists of the schema of chunk $\mathcal{D}$, the *metaChunk* $\mathcal{M}$ to scan and the predicate tree $\mathcal{P}$. The algorithm is a process traversing $\mathcal{M}$ with $\mathcal{P}$. The output of the operator is a boolean value indicating whether the chunk contains the valid data, which means the values specified by the query can be found in corresponding *dataChunk*.

The traversal of the *metaChunk* is conducted from the root node of $\mathcal{P}$ in a Depth-First Search (DFS) manner. During the traversal, different actions are invoked based on the type of the traversed predicate node. For single predicate node, the validation on whether the existing range of its corresponding dimension d contains the given values specified by the query is processed. Take Figure 3 as an example, the validation on Node C succeeds once the *metaChunk* reflects that dimension "city" contains "LA" or "SF" for the corresponding *dataChunk* (Such information is recorded in the "fingers" for WordMetaField in the storage of *metaChunk* in Figure 2). For the joint predicate nodes, all child nodes whose type is *Single* are firstly visited and validated on the corresponding dimensions. Then, other joint predicate child nodes are traversed recursively until the validation on the entire tree is done. For instance, in Figure 3, the traversal on Node A firstly starts on Node C and then moves to Node B.

By firstly checking the single predicate child nodes of the joint predicate nodes, the cost of the validation is pruned since the traversal can be terminated early. For example, the validation on Node A in Figure 3, whose type is *And*, can return "false" immediately once the validation on Node C fails. Hence, the validation cost of Node B is pruned as the outermost logical operand of Node A is *and*. Similarly, the validation on Node B returns "true" immediately once the validation on Node D succeeds, and the cost of checking the other node of Node B is eliminated. To further reduce the traversal cost on the predicate tree, the same early termination is also applied for the recursive validation on joint predicate child nodes, where the result is returned directly once the validation returns wanted results for either *Or* or *And* nodes.

It is worth mentioning that *Cool* always performs the predicate operators on time dimension first for multiple predicates on the same level of the tree. This is a pruning strategy in scanning to reduce query latency since many records can be quickly skipped. Due to the fact that the records for each user are sorted chronologically, those whose timestamp is out of the given time interval can be filtered out easily within at most two comparison operations – one is for the minimal value while the other one is for the maximal value – are needed. Such optimization is an effective solution for the data with time-dimension input and has been widely adapted by state-of-the-art OLAP systems such as Druid [15] and Pinot [32].

dataChunk Selector $\sigma^D_{\mathcal{P}}$. The *dataChunk* selector is used to find the matched data records in query processing.The input of the operator includes *dataChunk* $\mathcal{R}$, the number of records $\mathcal{N}$ and predicate tree $\mathcal{P}$. The output of the operator is a bitset B, whose length equals to the number of records the *dataChunk* contains. Each bit of B represents a position which a corresponding record holds in $\mathcal{R}$. When the bit is set (equals to 1), the record stored in the corresponding position in $\mathcal{R}$ is matched to the conditions contained in the predicate tree $\mathcal{P}$ and should be selected for later aggregation. On the contrary, the record is excluded in the aggregation when the bit is unset

(equals to 0).

Similar to *metaChunk* selector, the selector traverses the predicate tree from the root node and acts differently according to the type of the predicate nodes. For a single predicate node, the algorithm scans all the records in related dimension and sets the corresponding bit when the record matches the predicate. For a joint predicate node whose type is *Or*, the union of all the result bitsets of its child node is returned, calculated by recursively traversing the child nodes and scanning corresponding records to find the match. The intersection of all the result bitsets is returned for the joint predicate node whose type is *And*. Similar to *metaChunk* operator, the optimization, processing the predicates on time dimension with the highest priority, is adopted to prune the scanning records.

For cohort queries, we take advantage of the essential operators described in [6] and implement them with imperative adaption for the enhanced version of cohort analysis. Particularly, we change the semantics of birth selector and age selector to support our version of cohort analysis and leave cohort aggregator as it is in the original implementation.

Birth Selector σ^B_P. The birth selector is to capture the qualified users who pass the validation on the predicates in their event dimension. The birth selector accepts the *metaChunk*, the *dataChunk* and the predicate tree $\mathcal{P}$ as input. The selector first checks *metaChunk* whether the events in birth event sequence exist in the scanning chunk and then process the selection on *dataChunk*.

Age Selector $\sigma^G_{P,\mathcal{B}}$. The age selector is used to retrieve the age events for the users who satisfy the given birth actions. The input of the selector contains the *dataChunk*, the predicate tree $\mathcal{P}$, and the bitset $\mathcal{B}$, marking whether the user can be selected for aggregation in age. Despite the age delimiter, which can be only specified in time dimension supported in [6], we extend this operator to support event-based age delimiters. Intuitively, the boundary between two consecutive ages can be decided by either a specific event or a fixed time interval.

Cohort Aggregator $\gamma^C_{\mathcal{B},m_f}$. The records that pass both the birth selector and the age selector are grouped by different user cohorts and aggregated on age. The operator receives the bitset $\mathcal{B}$, which indicates the position the valid records hold in the cublet, and the metrics m_f, which is the user-defined measurement over the field f. The aggregator scans all valid records and aggregates the measurement into the aggregate value based on the name of the cohort and the age. Thus, a set of triads, consisting of the name of the cohort, the age and the corresponding aggregate value, is output.

4) Processing Flow:

OLAP Queries. To process OLAP queries, the following steps are conducted in the executor.

(1) Receive the execution plan produced by the planner.
(2) Fetch a cublet from the specified data source.
(3) Run metaChunk selector $\sigma^M_{\mathcal{P}}$ to scan *metaChunk*, checking whether the cublet contains the candidate values.
(4) Go to Step (2) if there are no values matched in *metaChunk*.
(5) Run dataChunk selector $\sigma^D_{\mathcal{P}}$ to scan *dataChunk* locating matched records.
(6) Call aggregators on the scanning result for *dataChunk* and collect the aggregation values into groups.
(7) Repeat Step (2) - (6) until all cublets are processed.
(8) Call a compressor to store the aggregate results if necessary.
(9) Output the final results.

When processing OLAP queries, the executor first gets a plan including the predicate tree $\mathcal{P}$ from the planner (Step 1). Then, the executor begins to scan the cublets in the data source. Recall that in Figure 2, each cublet contains a *metaChunk*, recording the values of each field appearing in corresponding cublet, the executor makes use of this information to prune the unnecessary data scanned by checking whether such values fulfil the predicate plan as presented in Section III-C3. The executor skips the current cublet and starts to scan the next chunk (Step 4) if the output of *metaChunk* selector $\sigma^M_{\mathcal{P}}$ is false and continues to scan the current *dataChunk* (Step 5) if the output is true. The scan of the *dataChunk* is processed via the selection operator $\sigma^D_{\mathcal{P}}$. In step 6, the executor aggregates the metrics specified in the query based on the scanning results of Step 5 and updates the aggregate values in an intermediate group map, where the keys are generated by group names. After all cublets are processed, the executor flushes the aggregate result into the storage if the query specified and renders the result as the output.

Cohort Query Processing. Recall that we adapt the birth selection operator, the age selection operator and the cohort aggregation operator from Cohana [6], and integrates them into *Cool*. The entire process flow is described as follows.

(1) Receive the execution plan produced by the planner.
(2) Fetch a cublet from the specified data source.
(3) Run birth selector σ^B_P to scan *metaChunk* checking whether the cublets contain the events in the birth event sequence.
(4) Go to Step (2) if there is no event matched in *metaChunk*.
(5) Run birth selector σ^B_P to scan *dataChunk* locating the birth users.
(6) Run age selector $\sigma^G_{P,\mathcal{B}}$ to calculate ages.
(7) Run cohort aggregator $\gamma^C_{\mathcal{B},m}$ to aggregate different cohorts from the users.
(8) Repeat Step (2) - (7) until all cublets are processed.
(9) Call a compressor to store the results if necessary.
(10) Output the cohort aggregation results.

As mentioned in Section III-A, we further enhance the functionality of cohort analysis by introducing *birth event sequence* and *reference cohort* compared to Cohana [6]. The motivation of introducing such enhancement is the limitation on the outdated definition. On the one hand, the original cohort analysis cannot be applied in many cases since only single birth event is supported and used within the birth selection operator. *Cool* offers the selection based on a sequence of birth events so that the definition of cohorts can be relaxed in order to support a wider range of queries. In this way, the birth

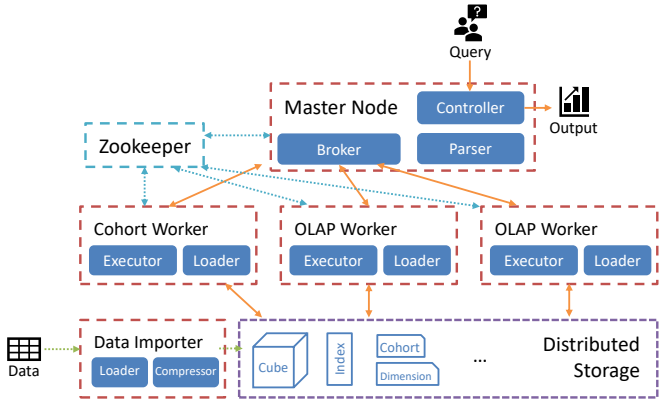


Fig. 4: *Cool* Distributed System Architecture

selection now can be applied on both event field and time field and the users who have done the sequence of such events can be easily detected.

On the other hand, it is impossible to conduct nested cohort analysis in Cohana, which means the system cannot process a successive cohort query based on the result of a precedent cohort query. *Cool* addresses this problem by storing the matched records of the precedent cohort query and treat them as a new data source called "reference cohort". Consequently, the system is now able to run complex queries such as checking whether the treatment remains effective after the readmission for the user cohort "130-160 mg/dL" in Table I.

To summarize, *Cool* gains the performance boost mainly from three optimizations. Firstly, *Cool* can use less memory to represent the data compared to traditional systems due to the sophisticated storage design. That is, *Cool* scans more data records than other systems in each disk fetch and therefore encounters less slow disk fetches for the same dataset. Secondly, the tuning strategies in query processing further reduce the number of records to scan, leading to competitive performance gains. The system can skip the scanning of the cublets without the queried values by *metaChunk* selection. Meanwhile, the execution of predicates further cut down the scanning cost as the process can be terminated early as described previously. Thirdly, the implementation of native cohort operators can accelerate cohort query at scale since the expensive join operation is eliminated as compared to the implementation in conventional systems.

IV. Distributed System Architecture

In this section, we extend *Cool* for distributed processing in order to serve big data needs without compromising the ability and efficiency in supporting both OLAP and cohort queries.

A. System Architecture and Query Processing

The architecture of distributed *Cool* is depicted in Figure 4, where each worker stands for a node in a distributed network. Naturally, a distributed storage is used instead of the local storage and besides all the components in single-node architecture, an additional broker is introduced. With the help of Zookeeper, the broker is responsible for the allocation of tasks among different workers, and the message passing for cross-node communication is via HTTP protocol. The data injection procedure now is fully assigned to a data importer, which is individually operated out of the processing system and can be called automatically according to the business needs by the developers. A compressor and a loader are embedded into the importer to assist data injection process. It is worth mentioning that the storage of the query results is also assigned to the importer, which is called after the query processing.

The query format for multi-node *Cool* is the same as the single-node *Cool*, which means a parser is needed to parse the query. However, the producer of the query plan differs. The broker, instead of the planner in single-node version, is responsible for the generation of the query plan in multi-node *Cool*. Since the data is divided into cublets as described in Section III-A, the query plan now contains different cublet sets to scan for each worker. After the generation of the query plan, the broker distributes it to idle worker nodes. As soon as each worker node receives the query plan, it starts to process the query with operators similar to the single-node version. The processing results produced by the worker nodes are stored in a temporary directory of the distributed storage, indexed by the id of the worker node. Once the processing is complete, each worker node individually invokes the broker to check whether all the participated workers have completed their job. The final results are merged by the broker and output by the controller in the master node.

B. Load Balancing and Fault Tolerance

The multi-node *Cool* supports basic load balancing in terms of task scheduling and necessary fault tolerance to node failure. Compared to the large size of cublets processed by multiple worker nodes, the size of the results is much smaller and hence the combination of partial results does not affect the overall performance severely although it is only conducted by the broker. Therefore, we use Zookeeper to maintain the processing status of all workers, which can be accessed by the broker when assigning query processing tasks and merging results. Moreover, a scheduling queue is implemented in *Cool* and the tasks assigned for the workers are pushed into the queue instead of directly flushed when the worker is busy. The broker can re-assign the idle workers with unprocessed task from the queue and therefore eliminates meaningless wait in query processing as much as possible.

Despite the node status mentioned in the last paragraph, the address of the master node and the thread id of the broker are also recorded in the zookeeper. There exists a heartbeat detection in *Cool* to guarantee that the system can recover from the failures from either the master node or the worker node. If a worker node crashes, the broker in the master node notifies Zookeeper and marks its status as "lost". The processing task the crashed node works on is then transferred to other alive idle worker nodes via re-sending its query plan by the broker. If a master node crashes, the first idle worker node in the worker list automatically picks itself as the new master node and changes the corresponding keys in zookeeper. Meanwhile, the

TABLE II: Line of Code for the queries

loc: Query / System	Q1	Q2	Q3	Q4
Cool	19 lines	15 lines	11 lines	17 lines
SQL	27 lines	5 lines	6 lines	33 lines
Druid	-	-	13 lines	-

TABLE III: Dataset Statistics

Dataset	TPC-H			
	tiny	small	medium	large
#records	7.5M	15M	30M	60M
size	3.1G	6G	13G	25G

Dataset	MED			
	tiny	small	medium	large
#records	8.6M	17M	34M	69M
size	3.2G	6.4G	13G	26G

query results can be easily recovered since they are maintained in the distributed storage separately and referred by zookeeper.

V. Performance Evaluation

Cool has been designed and optimized as an integrated system to support both cube queries and emerging cohort queries. To the best of our knowledge, there is no similar system that we can use as a baseline to compare all the functionalities. We therefore select Apache Druid [15], one of the most popular event-based systems for OLAP-related query evaluation, and *MonetDB* [13], a fast columnar analytical database for cohort query evaluation. For comparison on distributed processing, we select *SparkSQL* and distributed *Druid*. The measurement of the system includes the query latency of queries, the compression ratio of the datasets, and the memory consumption in processing. We use scripts to check whether the output results from different systems are the same and present the average metric on ten runs for each case. The query result is ensured to be the same for all the candidates being benchmarked.

Two datasets are used in this experiment. The first dataset is from TPC-H benchmark generated by official scripts[1]. Using the script, we generate the records for two tables, i.e., *customer* and *orders*, for the queries. The second, a medical dataset denoted as Med, is generated randomly based on a real database schema from one of the public hospitals we have been working with. The important fields include the event, the timestamp and the treatment status and the institution where the patient resides. The benchmarking queries are defined as follows.

Q1. Cohort query: For MED dataset, find the number of patients from different medical institutions, who are put into the observation list before their operation and their treatment remains effective after the operation.

Q2. Iceberg query: For TPC-H dataset, find all countries in Europe area and its total amount of orders with priority equals to "2-HIGH" between January 1st, 1993 and December 31st, 1993.

Q3. Cube query: For TPC-H dataset, find different regions and its total amount of orders by month.

Q4. Composite query: For MED dataset, find the total number of patients who belong to the cohort of Q1 by user's district.

As listed, Q1 is a retention problem by counting the number of patients whose treatment retains effective, which can be used to evaluate the performance of *Cool* in terms of cohort analysis query processing. Q2 is an iceberg query containing multiple selection conditions, which is aimed to evaluate the efficiency of the database selectors implemented by *Cool*, while Q3 is a typical roll-up query over the time axis of a data cube (the minimal unit of time dimension is day), which is targeted on the cube queries. The purpose of using Q4 in the experiment is to validate the ability of the system in running composite queries.

A. Query Example and Usability

We first illustrate the queries submitted to *Cool*, namely Q1 and Q2, in Listing 1 and Listing 2. The full schema is omitted due to space constraint. To compare, line of code (loc) is collected for the three systems and the results can be referred to Table II, where '-' means that the query is not applicable for the system. As can be observed, *Cool* needs fewer lines of code for cohort query Q1 and composite query Q4 compared to SQL. Nonetheless, *Cool* requires more lines of code for iceberg query Q2 and cube query Q3. The outcome is natural and apparently *Cool* sacrifices certain usability to win the query expressiveness in supporting different query types in system level.

B. Single-node Benchmark

The three systems are run with an Ubuntu 14.04 server with 8G memory and 2-core CPU (2.2 GHz) in this experiment. We create four different sizes, i.e., tiny, small, medium and large, for both datasets. The detailed statistics on the datasets are described in Table III, where the number of records varied from 7.5 million to 69 million and the range of the actual size varies from 3.1 Gigabytes to 25 Gigabytes.

1) Query Latency:

We study the query latency of all the queries for the three systems and the results are shown in Figure 5.

For the first query (Q1), we benchmark *Cool* and *MonetDB*. Cohort query is not supported in *Druid* as it is impossible to select on birth event sequence upon the injected data[2] according to the official documents and hence it is omitted in this case. From the results shown in Figure 5(a), we can observe that *Cool* is generally faster than *MonetDB* in one order of magnitude with respect to the query latency. The largest gap appears in the large dataset, where the absolute values are 25.3 seconds for *MonetDB* and 1.5 seconds for *Cool*. The major reason for the gap is that *MonetDB* incurs more disk accesses during the processing compared to *Cool*. The frequent data swapping, from disk to memory undoubtedly slows down the system performance in terms of query latency.

[1]https://github.com/electrum/tpch-dbgen

[2]https://druid.apache.org/docs/latest/tutorials/tutorial-rollup.html

```
{
  "dataSource": "MED",
  "referName": "Q1_Result",
  "birthSelection":[{
    "dimension":"INSTITUTION",
    "values": ["NUS", "NHGP"],
    "birthActions": [
      { "value": "observation start", "order": 1 },
      { "value": "operating", "order": 2 }
    ]
  }],
  "ageSelection": [
    { "selectType": "set", "dimension": "TR_STATUS", "values": ["effective"] }
  ],
  "cohortBy": {
    "dimension": "INSTITUTION",  "ageInterval": "DAY",
      "cohortMetric": "Retention"
  }
}
```

Listing 1: Cohort query example (Q1)

```
{
  "dataSource": "tpc-h",
  "predicates": {
    "type": "and",
    "children": [
            { "type": "single", "dimension": "O_ORDERPRIORITY", "operator": "=",
              "values": ["2-HIGH"]},
            { "type": "single", "dimension": "R_NAME", "operator": "=",
              "values": ["EUROPE"]},
            { "type": "single", "dimension": "O_ORDERDATE", "operator": "in",
              "values": ["1993-01-01|1993-12-31"]}
        ]},
  "groupBy":[{ "dimension": "N_NAME" }],
  "aggregations":[{ "dimension": "O_TOTALPRICE", "operator": "COUNT" }]
}
```

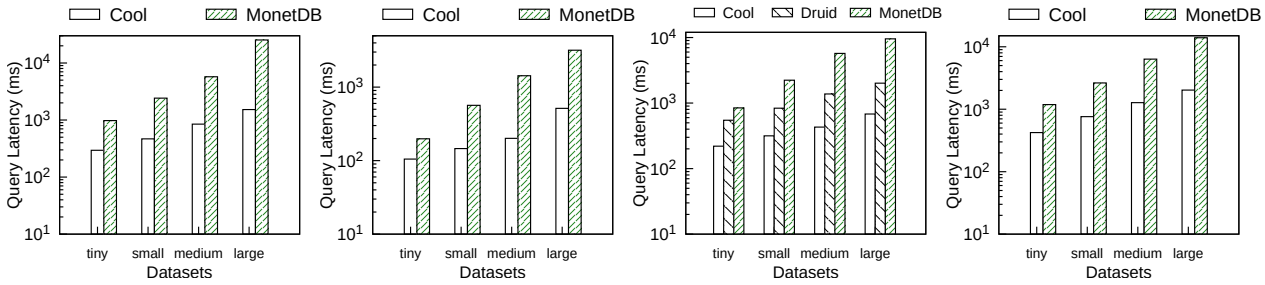
Listing 2: OLAP query example (Q2)

For the second query, we separate data injection into two steps to simulate the processing in a real-time manner [3]. The first portion of data, records between January 1st, 1993 and November 30th, 1993, is initially imported into the system. After the query processing is done on the first portion of data and the result is generated, the second portion of data, records between December 1st, 1993 and January 1st, 1994, is injected as if it is the real-time data produced from ETL tools. The system needs to update the existing results to answer the second query and the query latency of updates is accumulated and drawn in Figure 5(b). We implement this workflow in SQL for *MonetDB* and use materialized view to store the output of the results on the first portion of data and also the final results. The time used to update the materialized view is therefore collected as the query latency of Q2 in this setting. *Druid* is absent in this test since the system cannot answer such kind of real-time queries.

As can be observed in Figure 5(b), *Cool* outperforms *MonetDB* by almost one order of magnitude in real-time settings. However, the gap between the two candidates becomes small when the size of the dataset grows. The reason for such outcome is the reduction of the records scanned in the system. *MonetDB* incurs less disk accesses since it only needs to deal with the incremental dataset and update the existing results. The decrease of the absolute value for the query latency can also indicate this trend, where *MonetDB* only uses 198 milliseconds in Figure 5(b) on tiny dataset compared to 979 milliseconds on the same dataset for Q1.

Since Q3 is a typical roll-up query over a data cube with time dimension, we can easily implement the query in *Druid*. As can be seen in Figure 5(c), the query latency of *Cool* is almost one third of the query latency of *Druid*. The reason which causes the differences between the two systems stems from the implementation of the precomputation of the dimensions in data cube. Although we use precomputation on time dimension for *Druid* by setting the granularity to day when injecting the dataset, it is not as efficient as *Cool*'s bitset

[3]The test on the entire dataset produces similar results to Figure 5(a).



(a) Q1 (b) Q2 (c) Q3 (d) Q4

Fig. 5: Query Latency

implementation since *Druid* has more complex management tasks for the data segments.

For Q4, we directly measure the time period from the query submission to the result receiving. As can be seen, the difference value between the two systems becomes larger compared to the gap in either Figure 5(a) or Figure 5(d). This is caused by the fact that *Cool* can directly derive the results of Q4 from the results on the precedent query, namely Q1, while *MonetDB* cannot directly make use of the results of Q1 due to the heterogeneity of the input and output formats of the two queries. In practice, *MonetDB* has to run a composite query which meets both the criteria of Q1 and Q4.

2) Processing Memory:

We compare the memory consumption during query processing between *Cool* and *MonetDB* for Q1 and compare the same metric between *Cool* and *Druid* for Q3. The results are shown in Figure 6(a) and Figure 6(b), respectively.

As can be seen in Figure 6(a), *Cool* performs better than *MonetDB* for all the datasets; Especially in the tiny dataset, *Cool* consumes only 232 megabytes of memory while *MonetDB* takes 1999 megabytes of memory. The reason is that *Cool* utilizes compression schemes with higher efficiency than *MonetDB* and incurs no additional memory costs to maintain the schema of both the tables and the dimensions. For Q3, it can be observed from Figure 6(b) that *Druid* consumes much more memory space than *Cool*. Even for the largest dataset, the memory used by *Druid* is still three times of that used by *Cool*. This is mainly due to two reasons. First, *Druid* needs to maintain auxiliary system components in order to manage the data segments, which is complex and incurs high overhead. However, *Cool* exploits a relatively simple storage hierarchy and therefore eliminates such extra costs. Second, *Druid* must map the data from disk to memory during query processing even for the tiny dataset while for *Cool*, due to the highly compressed cublet structures, it may keep the entire dataset in memory for small datasets, which further leads to the performance gap between the two systems.

C. Compression Ratio

To assess the compression efficiency for the sophisticated storage layout in Figure 2, we compare the size of the raw datasets and the size of the compressed datasets in this test case. We further make a breakdown analysis here by comparing the compressed datasets with and without the compression on match set enabled. The results are shown in Figure 7.

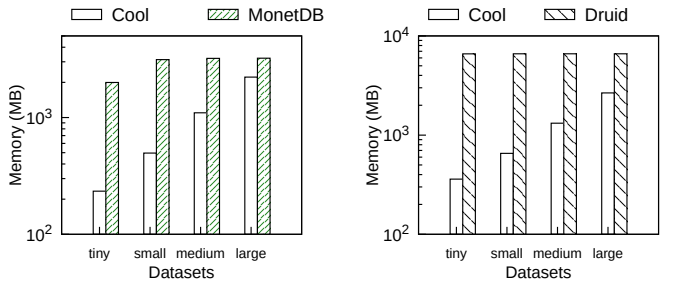
It can be inferred from the figure that the compression ratio of the raw data to our storage design is around 3. For the tiny dataset, *Cool* can even achieve a compression ratio of 3.8, where the absolute value is 0.779G compressed data vs. 3.1G raw data. Additionally, the RLE compression in match sets can achieve a compression ratio around 1.6 in most cases. The worst compression ratio is about 1.4 occurred in small dataset, where the size with match set compression enabled is 2.75G and the size with the compression disabled is 3.88G.

D. Multi-node Benchmark

For distributed processing benchmark, we employ 1 to 16 nodes to evaluate the system performance. Each node is equipped with Intel Xeon E5-2698 CPU (2.3 GHz) and 8GB memory. The operating system installed is Ubuntu 18.08. We generate 80 Gigabytes data for both datasets using the same scripts in single-node tests.

The results are shown in Figure 8. For *SparkSQL*, we complete the injection by converting the raw data into parquet partitions. Similar to single-node experiment, we also run Q3 on druid although the distributed version is employed. As can be seen from the figure, *Cool* runs one orders of magnitude faster than *SparkSQL* for cohort query (Q1). Specifically, *Cool* spends 7.03 seconds when processing with 16 nodes while *SparkSQL* incurs 328 seconds. Similar conclusion is also derived for the composite query (Q4). For iceberg queries (Q2), *Cool* achieves comparable performance to *SparkSQL*. As for 8-node case, *Cool* incurs 49.1 seconds while *SparkSQL* needs 45 seconds. The reason why the latency increases when the number of nodes changes from 8 to 16 for *SparkSQL* is that the merge of all the results, instead of the computation of each parquet partition, dominates the system performance. The advantage becomes more obvious for the composite query (Q4) since *Cool* can directly make use of the results of precedent query, namely Q1, while *SparkSQL* needs to run Q1 first before processing Q4. For cube queries (Q3), *Cool* achieves comparable latency to *SparkSQL* and performs slightly better than *Druid* with 1 worker node. For 16-node case, *Cool* can get 3x faster latency than the two systems.

SparkSQL achieves better scalability in iceberg query than cohort query and OLAP query. The execution time for *Spark-*



(a) Q1 (b) Q3

Fig. 6: Memory Consumption

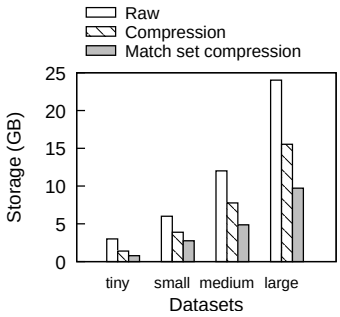
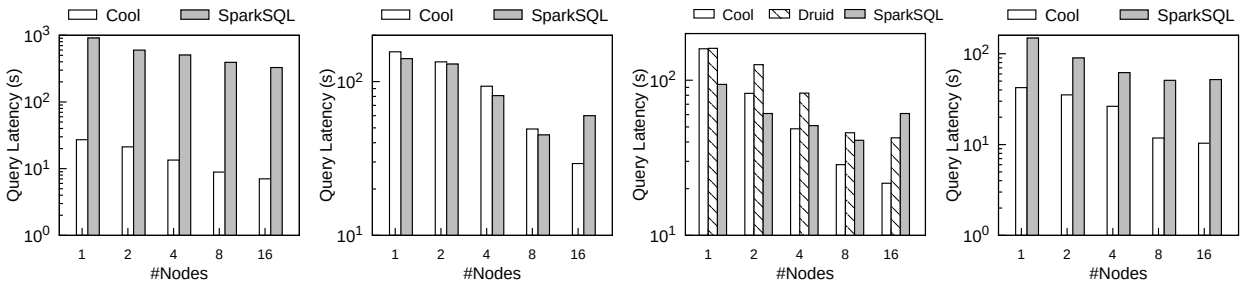


Fig. 7: Compression Efficiency



(a) Q1 (b) Q2 (c) Q3 (d) Q4

Fig. 8: Scalability

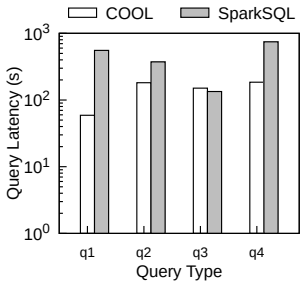


Fig. 9: Query Latency on Huge Dataset

SQL shrinks more than a half when the number of nodes increases from 4 to 8 in Figure 8(b) while the same measurement only drops around 40% with the same change. Similar trend can also be observed for *Cool*. The main cause of such trend is that cohort query requires a more complex aggregator than iceberg query, which is commonly done by a single node and therefore hinders the scalability.

We further extend the test on a huge dataset for *Cool* and *SparkSQL* to compare the query latency[4]. In this test, we run 16 worker nodes equipped with 32GB memory and 2.3 GHz CPU for the two system and the size of the two datasets is 800GB. The results for the four queries are shown in Figure 9. As can be observed, *Cool* has comparable query latency to *SparkSQL* for cube query (Q3) and runs slightly better than *SparkSQL* for iceberg query (Q2). Moreover, *Cool* is almost one order of magnitude faster than *SparkSQL* in terms of cohort query (Q1). The results of the four different queries indicate that *Cool* has equal query expressiveness to *SparkSQL* yet achieves better query performance.

[4]*Druid* fails the ingestion after we wait for three days in this test case.

TABLE IV: System Comparison in OLAP scenarios

System	Query Latency	Query Expressiveness	Real-time Ingestion Support	Distributed Processing Support
RDBMS	High	Very High	Not Typically	Not Typically
Cohana	Low	Low	No	No
Druid	Moderate	Moderate	Yes	Yes
Spark	Moderate	High	Yes	Yes
Cool	Low	High	Yes	Yes

E. Feature Comparison

We summarize the pros and cons of *Cool* as well as the systems tested in our experiments in Table IV. Druid, Spark and *Cool* all support real-time ingestion and distributed processing. However, as confirmed by our evaluation results, the query latency provided by *Cool* is lower than the other two systems. Besides, it is obvious that traditional RDBMS trades off the query performance for the highest query expressiveness while Cohana sacrifices the query expressiveness as much as possible for a superb query latency.

We additionally compare the query latency of Cohana and *Cool* to verify this statement. Due to the inability of running all the previous queries for Cohana, we adapt Q1 to a simpler query, namely "for MED dataset, find the number of patients from different medical institutions who are operated and the treatment remains effective after the operation". The result of this test shows that, in terms of query latency, Cohana is around 12% superior than *Cool* (292ms vs. 345ms) on single-node settings for the specific query, proving the efficiency of the system. Considering the fact that *Cool* performs the fastest in cohort query processing among other baselines with

higher or equivalent query expressiveness, this result shows that *Cool* trades off little performance in query processing for the significant enhancement in supporting wider range of query types compared to Cohana.

VI. Conclusions

We propose *Cool*, an online cohort analytical processing system, to support both cohort based and conventional data analytics. We present details of *Cool*, including the design principle, the architecture, the storage hierarchy and the execution framework, and conduct an extensive performance evaluation. Further, we extend the system to a distributed environment with sufficient load balancing and fault tolerance support. Despite providing the full functionality to support both OLAP queries and cohort queries, the experimental results show that *Cool* has superb performance in terms of query efficiency and memory consumption compared to *Druid* and *MonetDB*. For distributed settings, *Cool* outperforms *SparkSQL* and *Druid* by one order of magnitude in query latency.

Acknowledgment

This research is supported by Singapore Ministry of Education Academic Research Fund Tier 3 under MOE's official grant number MOE2017-T3-1-007. Gang Chen's work is supported by National Natural Science Foundation of China (Grant No. 61672455), and Natural Science Foundation of Zhejiang Province (Grant No. LY18F020005).

References

[1] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *SIGMOD Record*, vol. 26, no. 1, pp. 65–74, 1997.

[2] G. Colliat, "Olap, relational, and multidimensional database systems," *SIGMOD Record*, vol. 25, no. 3, pp. 64–69, 1996.

[3] M. De Rougemont and P. T. Cao, "Approximate answers to olap queries on streaming data warehouses," in *DOLAP*. ACM, 2012, pp. 121–128.

[4] "Additional cohort analysis example: Tableau software." [Online]. Available: https://kb.tableau.com/articles/howto/additional-cohort-analysis-example

[5] "Performing cohort analysis using mysql." [Online]. Available: https://chartio.com/resources/tutorials/performing-cohort-analysis-using-mysql/

[6] D. Jiang, Q. Cai, G. Chen, H. Jagadish, B. C. Ooi, K.-L. Tan, and A. K. Tung, "Cohort query processing," *VLDB*, vol. 10, no. 1, pp. 1–12, 2016.

[7] "Amplitude," https://amplitude.com.

[8] "Retention," https://mixpanel.com/retention/.

[9] "Rjmetrics," https://rjmetrics.com/.

[10] "Introduction to analytics: Funnel analysis." [Online]. Available: https://mixpanel.com/blog/2009/06/10/introduction-to-analytics-funnel-analysis/

[11] C. Zhan, M. Su, C. Wei, X. Peng, L. Lin, S. Wang, Z. Chen, F. Li, Y. Pan, F. Zheng *et al.*, "Analyticdb: Real-time olap database system at alibaba cloud," *VLDB*, vol. 12, no. 12, pp. 2059–2070, 2019.

[12] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan, "Amazon redshift and the case for simpler data warehouses," in *SIGMOD Record*. ACM, 2015, pp. 1917–1923.

[13] S. Manegold, P. A. Boncz, and M. L. Kersten, "Optimizing database architecture for the new bottleneck: Memory access," *The VLDB Journal*, vol. 9, no. 3, pp. 231–246, 2000.

[14] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs *et al.*, "Impala: A modern, open-source sql engine for hadoop." in *CIDR*, vol. 1, 2015, p. 9.

[15] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in *SIGMOD Record*. ACM, 2014, pp. 157–168.

[16] A. Hall, O. Bachmann, R. Büssow, S. Gănceanu, and M. Nunkesser, "Processing a trillion cells per mouse click," *VLDB*, vol. 5, no. 11, pp. 1436–1446, 2012.

[17] D. Lemire and O. Kaser, "Reordering columns for smaller indexes," *Information Sciences*, vol. 181, no. 12, pp. 2550–2570, 2011.

[18] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution," in *CIDR*, 2005, pp. 225–237.

[19] D. Bartholomew, "Mariadb vs. mysql," *Dostopano*, vol. 7, no. 10, p. 2014, 2012.

[20] M. N. Vora, "Hadoop-hbase for large-scale data," in *ICCSNT*, vol. 1. IEEE, 2011, pp. 601–605.

[21] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *TKDE*, vol. 27, no. 7, pp. 1920–1948, 2015.

[22] N. Shamgunov, "The memsql in-memory database system." in *IMDM@ VLDB*, 2014.

[23] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T.-H. Lee *et al.*, "Oracle database in-memory: A dual format in-memory database," in *ICDE*, 2015, pp. 1253–1258.

[24] A. Kemper and T. Neumann, "HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots," in *ICDE*, 2011, pp. 195–206.

[25] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient Transaction Processing in SAP HANA Database - The End of a Column Store Myth," in *SIGMOD Record*, 2012, pp. 731–742.

[26] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *VLDB*, vol. 2, no. 2, pp. 1626–1629, 2009.

[27] L.-Y. Ho, T.-H. Li, J.-J. Wu, and P. Liu, "Kylin: An efficient and scalable graph data processing system," in *IEEE BigData*. IEEE, 2013, pp. 193–198.

[28] B. Arres, N. Kabbachi, and O. Boussaid, "Building olap cubes on a cloud computing environment with mapreduce," in *AICCSA*. IEEE, 2013, pp. 1–5.

[29] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *VLDB*, vol. 3, no. 1-2, pp. 330–339, 2010.

[30] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[31] W. M. Mason and S. Fienberg, *Cohort analysis in social research: Beyond the identification problem*. Springer Science & Business Media, 2012.

[32] J.-F. Im, K. Gopalakrishna, S. Subramaniam, M. Shrivastava, A. Tumbde, X. Jiang, J. Dai, S. Lee, N. Pawar, J. Li *et al.*, "Pinot: Realtime olap for 530 million users," in *SIGMOD Record*. ACM, 2018, pp. 583–594.

[33] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman, "Computing iceberg queries efficiently," in *VLDB*, 1998.

[34] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data mining and knowledge discovery*, vol. 1, no. 1, pp. 29–53, 1997.

[35] D. Xin, J. Han, X. Li, and B. W. Wah, "Star-cubing: Computing iceberg cubes by top-down and bottom-up integration," in *VLDB*. VLDB Endowment, 2003, pp. 476–487.

[36] W. Wei, F. Jianlin, L. Hongjun, and J. X. Yu, "Condensed cube: an effective approach to reducing data cube size," in *ICDE*, 2002, pp. 155–165.

[37] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *SOSP*. ACM, 2013, pp. 423–438.

[38] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: fast data analysis using coarse-grained distributed memory," in *SIGMOD Record*. ACM, 2012, pp. 689–692.

[39] Z. Xie, Q. Cai, F. He, G. Y. Ooi, W. Huang, and B. C. Ooi, "Cohort analysis with ease," in *SIGMOD*, 2018, pp. 1737–1740.

[40] Q. Cai, Z. Xie, M. Zhang, G. Chen, H. Jagadish, and B. C. Ooi, "Effective temporal dependence discovery in time series data," *VLDB*, vol. 11, no. 8, pp. 893–905, 2018.