Cohort Query Processing

Dawei Jiang
National University of Singapore
jiangdw@comp.nus.edu.sg

Qingchao Cai
National University of Singapore
caiqc@comp.nus.edu.sg

Gang Chen
Zhejiang University
cg@zju.edu.cn

Beng Chin Ooi
National University of Singapore
ooibc@comp.nus.edu.sg

Kian-Lee Tan
National University of Singapore
tankl@comp.nus.edu.sg

Anthony K. H. Tung
National University of Singapore
atung@comp.nus.edu.sg

arXiv:1601.00182v2 [cs.DB] 5 Jan 2016

ABSTRACT

Modern Internet applications such as websites and mobile games produce a large amount of activity data representing information associated with user actions such as login or online purchases. Cohort analysis, originated from Social Science, is a powerful data exploration technique for finding unusual user behavior trends in large activity datasets using the concept of *cohort*. This paper presents the design and implementation of database support for cohort analysis. We introduce an extended relational data model for representing a collection of activity data as an activity relation, and define a set of cohort operators on the activity relations, for composing cohort queries. To evaluate a cohort query, we present three schemes: a SQL based approach which translates a cohort query into a set of SQL statements for execution, a materialized view approach which materializes birth activity tuples to speed up SQL execution, and a new cohort query evaluation scheme specially designed for cohort query processing. We implement the first two schemes on MySQL and MonetDB respectively and develop a prototype of our own cohort query engine, COHANA, for the third scheme. An extensive experimental evaluation shows that the performance of the proposed cohort query evaluation scheme is up to three orders of magnitude faster than the performance of the two SQL based schemes.

1. INTRODUCTION

E-commerce websites and mobile gaming apps often accumulate a huge amount of activity data representing information that are associated with user actions such as register, login and online purchases. Such activity data are often tabulated to provide insight into the behavior of the users in order to increase sales and ensure user retention. To illustrate, Table 1 shows some samples of a real dataset containing the information of 30M user activities collected by a mobile game. Each tuple of this table represents a user action and its associated information. For example, tuple t_1 means that in a role of `dwarf`, player 001 launched the game on 2013/05/19 in Australia.

To obtain insight based on such activity data, one obvious choice is to apply traditional SQL GroupBy operators. Still taking Table 1 as an example, if we want to look at users' shopping trend in terms of the `gold` (the virtual currency) spent, we may run the following SQL query Q_s.

```
SELECT Week, avg(Gold) as AvgSpent
FROM GameActions
WHERE Action = "shop"
GROUP BY Week(Time) as Week
```

Executing this query against the whole 30M dataset results in Table 2, where each tuple represents the average gold that users spent in shopping during a certain week. The result as shown in Table 2 seems to suggest a slight drop in shopping, and then a recovery. However, it is hard to explain the behavior and draw insight from this result. In particular, this result fails to reveal the true trend of in-game shopping, which as we shall see shortly, is the consequence of the **aging** and **social change** effects.

To obtain deeper insight into factors that affect the behavior of users, we will focus on the use of **cohort analysis** in this paper. Cohort analysis, originally introduced in Social Science, is a data analysis technique for assessing the effects of aging on human behavior in a changing society [8].

According to the social scientists, there are two major sources that can affect human behavior: 1) aging, i.e., people behave differently as they grow older and 2) social changes, i.e., people may change their behavior when the society they live in changes. In our in-game shopping example, players tend to buy more weapons in their initial game sessions than they do in later game sessions - this is the effect of aging. On the other hand, social change may also affect the players' shopping behavior, e.g., with new weapons being introduced in iterative game development, players may start to spend again in order to acquire these weapons.

With cohort analytics, we can study the trend of human behavior in three steps. First, users are assigned to different cohorts (each cohort is therefore a group of users) based on the period of time when users perform an action for the first time. This step is of vital importance in cohort analysis since its purpose is to exclude the effect of social changes. Social scientists think that people who were born in the same period will exhibit similar behavioral patterns. In our in-game shopping example, suppose we cohort players based on

Table 1: Mobile Game Activity Table

	Player	Time	Action	Role	Country	Gold
t_1	001	2013/05/19:1000	launch	dwarf	Australia	0
t_2	001	2013/05/20:0800	shop	dwarf	Australia	50
t_3	001	2013/05/20:1400	shop	dwarf	Australia	100
t_4	001	2013/05/21:1400	shop	assassin	Australia	50
t_5	001	2013/05/22:0900	fight	assassin	Australia	0
t_6	002	2013/05/20:0900	launch	wizard	United States	0
t_7	002	2013/05/21:1500	shop	wizard	United States	30
t_8	002	2013/05/22:1700	shop	wizard	United States	40
t_9	003	2013/05/20:1000	launch	bandit	China	0
t_{10}	003	2013/05/21:1000	fight	bandit	China	0

Table 2: Results of Q_s

Week	AvgSpent
2013-05-19	50
2013-05-26	45
2013-06-02	43
2013-06-09	42
2013-06-16	45

Table 3: Cohort Report for Shopping Trend

Cohort	Age (Weeks)				
	1	2	3	4	5
2013-05-19 (145)	52	31	18	12	5
2013-05-26 (130)	58	43	31	21	
2013-06-02 (135)	68	58	50		
2013-06-09 (140)	80	73			
2013-06-16 (126)	86				

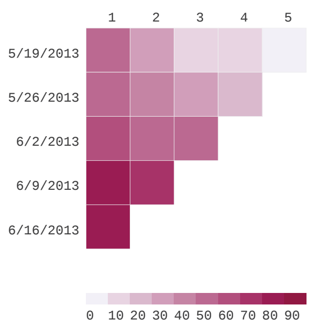

Figure 1: Launch Cohort Shopping Trend

Table 4: Cohort Report for Retention Trend

Cohort	Age (Weeks)				
	1	2	3	4	5
Australia (162)	160	103	78	42	26
United States (1815)	1812	1065	678	478	470
China (924)	912	630	457	234	125

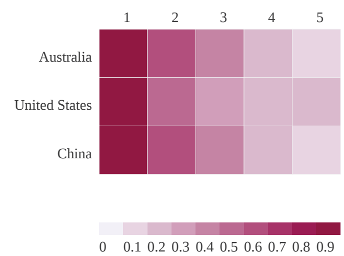

Figure 2: Country Cohort Retention Trend

the week when they first launched the game. Then, player 001 is assigned to 2013-05-19 launch cohort. In the second step, the activity tuples are also partitioned accordingly so that tuples of an user are assigned to the same cohort as the user. For our running example, all the shopping tuples are assigned to the cohorts to which their players belong. For instance, tuple t_2 is assigned to its player's 2013-05-19 launch cohort. Finally, in the third step, to capture the aging effect, the tuples of each cohort are further split into smaller sub-partitions based on **age** (time). The desired aggregate is then applied on each such sub-partition. In our in-game scenario, we organize the shopping tuples into partitions, each corresponding to a week's duration (i.e., *age*). In other words, for each shopping tuple in a cohort, we calculate the number of weeks (i.e., *age*) that have passed between the shopping time and the time the player first launched the game, and assign the tuple to the respective partition. Finally, we report average gold each cohort spent at different ages.

The result of our shopping trend cohort analysis is shown in Table 3 and visualized in Figure 1 in the form of a heatmap. In Figure 1, each row represents the shopping trend of a cohort, and each column captures the aging effect on the average expenditures of that cohort since its "birth" with respect to the time the game is first launched. For example, column k reports the average expenditures of the cohorts at the age of k weeks.

By looking at each row horizontally, we can see the aging effect, i.e., players spent more gold on buying weapons on their initial game sessions than their later game sessions. On the other hand, by comparing different rows (i.e., reading rows vertically), we can observe that the drop-off trend becomes better. This suggests that the iterative game development indeed improves the players' gaming experience as they seemed to be spending more gold on buying weapons to win battles.

Moreover, the analysis result can be used as a training dataset for other analytical techniques. For example, we can regard the age as an independent variable and the average gold players spent as a dependent variable. Thus, for each cohort, we can train a regression model to predict the average amount of gold that cohort will spend in subsequent weeks. Such information can be further utilized by a recommendation engine for recommending weapons of similar price to that cohort. The ability to pipeline analytical results to downstream analytical algorithms makes cohort analysis an interesting and powerful tool for transforming raw activity data from Table 1 into actionable insights such as trend analysis and/or recommendation.

While the classic cohort analysis we presented so far has been very useful in many analytic applications such as retention analysis [1], we hope to further widen its application in this paper. One limitation of classical cohort analysis is that time is the only dimension which is used for identifying cohorts. This is natural in social science research where birth time is considered to be the key attribute that determines people's behavior. However, in many other applications, there are other factors leading to the similarity in people's behavior. For example, while players might start their first game session in different weeks (hence having different birth time), they may still share the similar gaming experiences if they play the same version of the game. On the other hand, players who perform the first game session in the same week or month (hence having the same birth time) may still behave differently since they may launch different versions of the game. Therefore, we intend to generalize cohort analysis so that it is able to group people by any attributes of interests. If such a generalization is possible, we would be able to perform, say, a country cohort retention analysis which reports the total number of active players in each country cohort since that cohort is born. The result of this generalized cohort analysis is shown in Table 4 and visualized in Figure 2.

This paper presents the design and implementation of database support for generalized cohort analysis. To our best knowledge, our paper is the first work to provide database query-style support for the generalized cohort analysis. We first introduce an extended relational data model which represents a collection of activity tuples as an activity relation, a special relation with at least three attributes, each within a particular domain. We further propose three operators to manipulate activity relations a birth selection operator $\sigma^b_{C,e}$ for selecting activity tuples of qualified users, an age selection operator $\sigma^g_{C,e}$ for selecting qualified age activity tuples while retaining all birth tuples[1], and a cohort aggregation operator $\gamma^c_{\mathcal{L},e,f_A}$ for aggregating activity tuples. Cohort analysis can thus be formulated as a cohort query which is an expression composed of the three cohort operators.

To evaluate a cohort query, we present three schemes. The first scheme (called SQL based approach) translates a cohort query into a set of SQL statements by translating each cohort operator into expressions composed of standard relational operators. This approach is easy to implement but is inefficient to execute since the translation involves multiple joins between temporary tables. The second evaluation scheme (called the materialized view based approach) employs materialized views to speed up query processing. However, this technique is not scalable due to the significant increase in the size of materialized views. Thus, we propose a new query processing scheme specially designed for cohort query processing. We implement the first two schemes on MySQL and MonetDB. For the third scheme, we build a new cohort query engine, COHANA (COHort ANAlytics), and equip it with various optimization techniques. Through extensive experiments, we show that COHANA runs up to three orders of magnitude faster than a row-oriented database (MySQL) and 3.4X∼20X faster than a column-oriented database (MonetDB), and demonstrate the effect of the proposed optimization techniques.

The rest of the paper is organized as follows: Section 2 presents the foundations of cohort analysis. Section 3 presents the SQL based and the materialized view based approaches for processing cohort queries. We also present the basic idea of sort-aware cohort algorithms in this section. In Section 4, we present COHANA, a prototype query engine which refines the sort-aware cohort algorithms by several query optimization techniques for cohort query processing. Section 5 reports the experimental results. We present related work in Section 6 and conclude this paper in Section 7.

[1]The definitions of birth tuples and age tuples will be given in Section 2.2.

2. COHORT ANALYSIS FOUNDATIONS

In this paper, the term *cohort* is used to refer to a number of individuals who have some common characteristic in performing a particular action for the first time, and we shall use this particular action and the attribute values of the common characteristics as an adjective to identify a cohort. For example, a group of users who perform the first `login` (the particular action) on `2015 January` (the common characteristic) is called the `2015 January login cohort`. Similarly, a number of customers who make their first purchase at the United States forms a United States purchase cohort. Broadly speaking, cohort analysis is a data exploration technique which inspects the variance of measures, e.g., revenue, retention and sales, of different cohorts since they were **born** from performing a particular action.

2.1 Modeling Activity Data

We employ a relational approach to model activity data. A collection of activity data is represented as an instance of an activity relation, a special relation where each tuple represents the information associated with a single user activity. We will also call an activity relation an activity table. In this paper, the two terms, i.e., activity relation and activity table are used interchangeably.

Formally, an activity table D is a relation with attributes $A_1, A_2, \ldots, A_n$ where $n \geq 3$. A_1 is a string uniquely identifying a user, A_3 is also a string, representing an action chosen from a pre-defined collection of actions, and A_2 records the time at which A_1 performed A_3. Every other attribute in D is a standard data cube style attribute, which can be a dimension attribute representing a user property or a measure attribute representing a numeric value associated with the activity tuple. Since A_1, A_2 and A_3 have specific semantics, we write them as A_u, A_t and A_e respectively. Furthermore, an activity table has a primary key constraint on (A_u, A_t, A_e). That is, each user i can only perform a specific action e once at each instant time. As exemplified in Table 1, the first three columns correspond to the user (A_u), timestamp (A_t) and action (A_e) attribute, respectively. Role and Country are dimension attributes which respectively specify the role and the country of player A_u when performing A_e at A_t. Following the two dimension attributes is Gold, a measure attribute representing the virtual currency that player A_u spent for this action. We shall continue to use Table 1 as our running example for describing each concept in cohort analysis.

2.2 Basic Concepts of Cohort Analysis

The core concepts of cohort analysis are **birth time**, which defines when to start measuring user behavior, and **age**, which defines the time period over which the metrics of interests are aggregated for each cohort. Given an action $e \in \mathrm{Dom}(A_e)$, the birth time of user i is the first time that i performed e or -1 if i never performed e, as shown in Definition 1. We call an action e, a birth action if e is used to define the birth time of users.

Definition 1. Given an activity table D, a time value $t^{i,e}$ is called the birth time of user i if and only if

$$t^{i,e} = \begin{cases} \min \pi_{A_t}(\sigma_{A_u=i \wedge A_e=e}(D)) & if \quad \sigma_{A_u=i \wedge A_e=e}(D) \neq \emptyset \\ -1 & otherwise \end{cases}$$

where π and σ are the standard projection and selection operators and $e \in \mathrm{Dom}(A_e)$.

Technically, any action $e \in \mathrm{Dom}(A_e)$ can be specified as a birth action. However, in real applications, we often use an action such as "Signup" or "Purchase" which triggers a long sequence of user activities as a birth action. We now define the birth activity tuple $d^{i,e}$:

Definition 2. Given an activity table D, a tuple $d^{i,e} \in D$ is called the birth activity tuple of user i if and only if

$$d^{i,e}[A_u] = i \wedge d^{i,e}[A_t] = t^{i,e}$$

where e is a birth action.

Given the fact that (A_u, A_t, A_e) is the primary key of D, we conclude that for each user i, there is only one birth activity tuple of i in D for any birth action e if i performed e. Based on the birth time $t^{i,e}$, each activity tuple d of user i, i.e., $d \in D \wedge d[A_u] = i$, can be indexed by the time difference called age between $t^{i,e}$ and $d[A_t]$.

Definition 3. Given the birth time $t^{i,e}$, a numerical value g is called the age of user i in tuple $d \in D$, if and only if

$$d[A_u] = i \wedge g = d[A_t] - t^{i,e}$$

The concept of age is designed for specifying the time interval over which to aggregate the behavioral metric of a cohort. In cohort analysis, we always calculate the metric at positive ages. That is, if the age of an user in a tuple is negative, that tuple will be excluded from the final report. The activity tuple with a positive age is called an age activity tuple. Furthermore, in practical applications, the age g is normalized by a certain time unit such as a day, week or month. Without loss of generality, we assume that the granularity of g is a day. That is, we report each cohort's aggregated metric in terms of days since the users of that cohort were born.

Consider the example activity relation in Table 1. Suppose we use the action `launch` as the birth action. Then, the activity tuple t_1 is the birth tuple of player 001, and the birth time is 2013/05/19:1000. The activity tuple t_2 is an age tuple of player 001 performed at age 1.

2.3 Cohort Operators

We now present operations on activity tables. We introduce three new operators: two selection operators $\sigma^b_{C,e}$ and $\sigma^g_{C,e}$ for selecting qualified activity tuples to aggregate and a cohort aggregation operator $\gamma^c_{\mathcal{L},e,f_A}$ for calculating aggregations of the qualified activity tuples. These operators are our cohort versions of the standard relational selection (σ_C) and aggregation ($\gamma_{\mathcal{L},f_A}$) operators.

2.3.1 The $\sigma^b_{C,e}$ Operator

A birth selection operator $\sigma^b_{C,e}$ is used to obtain all activity tuples of qualified users whose birth activity tuples satisfy a specific condition C. The formal definition of $\sigma^b_{C,e}$ is given as follows.

Definition 4. Given an activity table D, the birth selection operator $\sigma^b_{C,e}$ is defined as

$$\sigma^b_{C,e}(D) = \{d \in D \mid i \leftarrow d[A_u] \wedge C(d^{i,e}) = \text{true}\}$$

where C is a propositional formula and e is a birth action.

Consider the activity relation D in Table 1. Suppose we want to derive an activity table from D which retains all activity tuples for users who were born from performing the launch action in Australia. This can be achieved using the following expression.

$$\sigma^b_{\text{Country=Australia,launch}}(D)$$

The result set of the above operation is $\{t_1, t_2, t_3, t_4, t_5\}$ which contains all activity tuples of the only qualified player 001.

2.3.2 The $\sigma^g_{C,e}$ Operator

Suppose we want to obtain an activity table from D which retains all birth activity tuples in D and only includes age activity tuples which satisfy a condition C. The age selection operator is designed for this purpose.

Definition 5. Given an activity table D, the age selection operator $\sigma^g_{C,e}$ is defined as

$$\begin{aligned}\sigma^g_{C,e}(D) = \{&d \in D | i \leftarrow d[A_u] \wedge \\ &((d[A_t] = t^{i,e}) \vee (d[A_t] > t^{i,e} \wedge C(d) = \text{true}))\}\end{aligned}$$

where C is a propositional formula and e is a birth action.

For example, suppose we choose the action `shop` as the birth action, and want to derive an activity table which retains all birth activity tuples in Table 1 but only includes age activity tuples which indicate users performing in-game shopping in all countries but China. The following expression can be used to obtain the desired activity table.

$$\sigma^g_{\text{Action=shop} \wedge \text{Country} \neq \text{China,shop}}(D)$$

The result set of the above selection operation is $\{t_2, t_3, t_4, t_7, t_8\}$ where t_2 is the birth activity tuple of player 001, t_3 and t_4 are the qualified age activity tuples of player 001, and t_7 and t_8 are respectively the birth activity tuple and the only qualified age activity tuple of player 002.

A common requirement in specifying $\sigma^g_{C,e}$ operation is that we often want to reference the attribute values of birth activity tuples in C. For example, given the birth action `shop`, we may want to select age activity tuples which indicate that users perform in-game shopping at the same location as their birth location. We introduce a `Birth()` function for this purpose. Given an attribute A, for any activity tuple d, the `Birth(A)` returns the value of attribute A in $d[A_u]$'s birth tuple:

$$\texttt{Birth}(A) = d^{i,e}[A]$$

where $i = d[A_u]$ and e is the birth action.

In our running example, suppose that users were born from their first performing shopping, and that we want to obtain an activity table which retains all birth activity tuples but only include age activity tuples which indicate that players performed shopping in the same role as they were born. The following expression can be used to fulfill this requirement.

$$\sigma^g_{\text{Role=}\texttt{Birth}\text{(Role),shop}}(D)$$

The result set of the above operation is $\{t_2, t_3, t_7, t_8\}$ where t_2 and t_7 are the birth tuples of player 001 and player 002, respectively, and t_3 and t_8 are the qualified age activity tuples of player 001 and player 002, respectively.

2.3.3 The $\gamma^c_{\mathcal{L},e,f_A}$ Operator

We now present the cohort aggregation operator $\gamma^c_{\mathcal{L},e,f_A}$ which aggregates over activity tuples. Briefly speaking, the $\gamma^c_{\mathcal{L},e,f_A}$ operator performs aggregation in two steps. First, it divides users into different cohorts based on the projection of users' birth activity tuples onto a specified cohort attribute set, and we call this step the cohort operation. Then, all activity tuples (including both the birth and the activity tuples) are assigned to the cohorts to which the users belong for aggregation. We shall elaborate on these two steps one by one.

The cohort operation is based on a cohort attribute set $\mathcal{L}$. Formally, given an activity table D with its attribute set $\mathcal{A} = \{A_u, A_t, A_e, A_1 \ldots, A_n\}$ and a birth action e, we pick up a cohort attribute set $\mathcal{L} \subset \mathcal{A}$ such that $\mathcal{L} \cap \{A_u, A_e\} = \emptyset$ and assign each user i to a cohort c specified by $d^{i,e}[\mathcal{L}]$. Consider the activity relation in Table 1. Given the birth action `launch` and the cohort attribute set $\mathcal{L}$={Country}, player 001 is thus assigned to the Australia launch cohort, player 002 is assigned to the United States launch cohort and player 003 is assigned to the China launch cohort. Similarly, if we choose $\mathcal{L}$={`Day`(Time)}, where the `Day`() function returns the date part of the Time attribute, as the cohort attribute set, for the same launch birth action, player 001 is assigned to the 2013/05/19 launch cohort and player 002 and player 003 are assigned to the 2013/05/20 launch cohort. After assigning users to cohorts, we thereafter divide activity tuples into cohorts accordingly. Given a cohort attribute set $\mathcal{L}$ and a birth action e, an activity tuple d is assigned to the same cohort c as is $d[A_u]$, i.e., $d^{i,e}[\mathcal{L}]$. The $\gamma^c_{\mathcal{L},e,f}$ operator is formally presented in Definition 6.

Definition 6. Given an activity table D, the cohort aggregation operator $\gamma^c_{\mathcal{L},e,f_A}$ is defined as

$$\begin{aligned}\gamma^c_{\mathcal{L},e,f_A}(D) =\{&(d_{\mathcal{L}}, g, s, m)| \\ &D_g \leftarrow \{(d,g)|d \in D \wedge i \leftarrow d[A_u] \wedge g = d[A_t] - t^{i,e}\} \\ &\wedge (d_{\mathcal{L}}, g) \in \pi_{\mathcal{L},g}(D_g) \\ &\wedge s = \text{count}(\pi_{A_u}\sigma_{d_g[\mathcal{L}]=d_{\mathcal{L}}}(D_g)) \\ &\wedge m = f_A(\sigma_{d_g[\mathcal{L}]=d_{\mathcal{L}} \wedge d_g[g]=g \wedge g>0}(D_g))\end{aligned}$$

where $\mathcal{L}$ is a cohort attributes set, e is a birth action and f_A is a standard aggregation function with respect to the attribute A.

In summary, the cohort aggregation operator takes an activity table D as input and produces a normal relational table R as output. Each row in the output table R consists of four parts $(d_{\mathcal{L}}, g, s, m)$, where $d_{\mathcal{L}}$ is a projected tuple on the cohort attributes set $\mathcal{L}$ representing the cohort, g is the age, i.e., the time point that we report the aggregates, s is the cohort size, i.e., the number of users in the cohort specified by $d_{\mathcal{L}}$, and m is the aggregated measure produced by the aggregate function f_A. Note that we only apply f_A on age activity tuples with $g > 0$.

2.3.4 Discussions on Cohort Operators

We note that the two selection operators, $\sigma^b_{C,e}$ and $\sigma^g_{C,e}$, are commutative if they involve the same birth action[2].

$$\sigma^b_{C,e}\sigma^g_{C,e}(D) = \sigma^g_{C,e}\sigma^b_{C,e}(D) \tag{1}$$

Leveraging this property, we present in Section 4 an optimization that pushes down the selection of birth tuples, i.e., birth selection operator, in evaluating cohort queries.

We can also check that $\sigma^b_{C,e}$ and $\sigma^g_{C,e}$ are operationally closed since they take an activity table as input and produce an activity table as output. However, the $\gamma^c_{\mathcal{L},e,f_A}$ operator is not operationally closed since it produces a normal relational table as output. Technically, it is easy to make $\gamma^c_{\mathcal{L},e,f_A}$ to be operationally closed. We just need to treat each cohort in the output table R of $\gamma^c_{\mathcal{L},e,f_A}$ as a giant user and extend each output tuple $(d_{\mathcal{L}}, g, s, m)$ to $(a_u, g, e, d_{\mathcal{L}}, s, m)$ where a_u is a string representation of $d_{\mathcal{L}}$, i.e., the concatenation of all attribute values in $d_{\mathcal{L}}$, and e is the birth action parameter of $\gamma^c_{\mathcal{L},e,f_A}$. We can check that (a_u, g, e) satisfies our (A_u, A_t, A_e) primary key constraint. Making the output of $\gamma^c_{\mathcal{L},e,f_A}$ as an activity table enables query chaining, that is, the output of a cohort query can be fed into another cohort query for further processing. However, we have not found any practical applications of this query chain since treating each cohort as a single giant user is quite strange and there is only one action, i.e., e, in the result set. The common scenario is that the output of a cohort query is further processed by a standard relational query such as selecting certain cohorts for comparison. Therefore, making $\gamma^c_{\mathcal{L},e,f_A}$ produces a normal relational table is sufficient.

2.4 The Cohort Query

Given an activity table D and operators $\sigma^b_{C,e}$, $\sigma^g_{C,e}$, $\pi_{\mathcal{L}}$, ρ [3] and $\gamma^c_{\mathcal{L},e,f_A}$, a cohort query $Q : D \rightarrow R$ is an expression of composition of those operators that takes D as input and produces a relation R as output, during which the following constraints are satisfied. 1) the same birth action e is used for all cohort operators in Q, and 2) the primary key (A_u, A_t, A_e) is always in the projection attribute set of $\pi_{\mathcal{L}}$, i.e., $\mathcal{L} \supseteq \{A_u, A_t, A_e\}$.[4] The first constraint ensures that the cohort query is meaningful, since it does not make sense to compare cohorts born from different birth actions. The second constraint ensures that the standard projection operator defined for a normal relation can work well with cohort operators by producing an activity table. Following is a cohort query regarding the shopping trend in our running example expressed with the proposed cohort operators.

EXAMPLE 1. *Given the launch birth action, report the total gold that country launch cohorts spent since they were born in the role of dwarf.*

$$Q_1 = \gamma^c_{\mathcal{L},e,\text{sum}_{Gold}}\sigma^g_{Action=shop,e}\sigma^b_{Role=dwarf,e}(D)$$

where $\mathcal{L} = \{Country\}$ *and* $e =$ *"launch".*

We do now allow standard selection operator σ_C and binary operations in the cohort query. The reason for rejecting σ_C is that the operator may produce *dangling users* in the result set. A dangling user is a user who only has age activity tuples but does not have the birth activity tuple in the

[2] In fact, , using the same birth action in $\sigma^b_{C,e}$ and $\sigma^g_{C,e}$ is a must for a cohort query to be meaningful.

[3] The renaming operator in relational algebra.

[4] Note that $\mathcal{L}$ is not the cohort attribute set here.

result set of an operation. For example, consider the activity relation in Table 1. Suppose we allow σ_C in the cohort query. One may specify the operation $\sigma_{\text{Time}>2013/05/22:0000}$. The birth tuple, i.e., t_1, is removed by the selection, and the result set of this selection operation is $\{t_5, t_8\}$, both of which are age tuples. The result activity relation is therefore corrupted in the sense that we cannot allocate player 001 to a cohort (since the birth activity tuple of that user is missing) and aggregate age activity tuples of that user anymore.

Binary operations such as join, intersection also suffers from the dangling users problem. To include them in cohort queries, these operators need to be redefined just as we do for selection operator. We therefore leave these extensions as future work.

From another point of view, the rejection of join operation in a cohort query may not be a big issue. In real applications, we can usually (if not always) pre-join all related tables into a single big wide table for analysis. In fact, this wide table scheme is very popular and is a preferred storage format for analytical databases in recent research [12].

3. ADDRESSING COHORT QUERIES

This section presents three schemes for the evaluation of cohort queries: SQL based approach, materialized view based approach and our proposed cohort algorithms. The first two schemes evaluate a cohort query by translating a cohort query into a set of SQL statements for execution. These two evaluation schemes are easy to implement, but are either inefficient or not scalable. we therefore develop a new cohort query processing scheme to achieve both efficiency and scalability. We present the basic algorithms of our own cohort query processing scheme in this section, and discuss in Section 4 the implementation of these algorithms as well as optimizations.

3.1 SQL based Approach

We now present the translation of each cohort operator into an expression of relational operators. Given a birth action e, we first derive a birth time table R^e that stores the birth time of users.

$$R^e = \gamma_{A_u, \min(A_t)\rightarrow A_t^b}\sigma_{A_e=e}(D)$$

where D is the input activity table and γ is the standard aggregation operator. For each user i in D, R^e stores i and its birth time $t^{i,e}$ with respect to the given birth action e.

With R^e, we can evaluate the birth selection operator $\sigma^b_{C,e}$ according to the following expressions.

$$T \leftarrow R^e \underset{R^e.A_u=D.A_u}{\bowtie} D \tag{2}$$

$$U \leftarrow \pi_{A_u}\sigma_{A_t=A_t^b\wedge C}(T) \tag{3}$$

$$\sigma^b_{C,e}(D) \leftarrow D \underset{D.A_u=U.A_u}{\bowtie} U \tag{4}$$

To evaluate $\sigma^b_{C,e}$, we first produce a temporary table T by joining the input activity table D with the birth time table R^e on A_u attribute (expression (2)). Then, we collect all qualified users in a temporary table U by first applying selection condition C on birth tuples of T (expression (3)) and then projecting the results on A_u. Finally, we join U and D and produce the final resultant activity table (expression (4)).

Compared with $\sigma^b_{C,e}$, the evaluation of $\sigma^g_{C,e}$ is similar but a little complicated. To perform $\sigma^g_{C,e}$, we first inspect the condition C to collect each attribute A_i for which the `Birth` function is given, and denote the set of collected attributes by $\mathcal{L}^b = \cup A_i$. Then the following expressions can be used to evaluate $\sigma^g_{C,e}$:

$$T \leftarrow R^e \underset{R^e.A_u=D.A_u}{\bowtie} D \tag{5}$$

$$U \leftarrow \pi_{A_u,\mathcal{L}^b\rightarrow\mathcal{L}^{b'}}\sigma_{A_t=A_t^b}(T) \tag{6}$$

$$\sigma^g_{C,e}(D) \leftarrow \pi_{\mathcal{A}}\sigma_{C^b\vee A_t=A_t^b}(D \underset{D.A_u=U.A_u}{\bowtie} U) \tag{7}$$

where $\mathcal{A}$ is the attribute set of D, the condition C^b in expression (7) is derived from C by replacing every `Birth` (A_i) function with A_i', a renamed attribute of A_i, in $\mathcal{L}^{b'}$. For example, suppose we want to evaluate the operation $\sigma^g_{\text{Time}=\texttt{Birth}(\text{Time}),e}$. First, we collect the attribute in `Birth()` function and obtain $\mathcal{L}^b = \{\text{Time}\}$. Suppose we rename the Time attribute in $\mathcal{L}^b$ to BirthTime in expression (6). Then, the condition C^b in expression (7) would be Time = BirthTime.

The evaluation of $\sigma^g_{C,e}$ consists of three steps. First, we derive a temporary table T by joining D and R^e on A_u (expression (5)). Second, we derive a temporary table U by retrieving all birth activity tuples from T and then projecting the results on $(A_u, \mathcal{L}^{b'})$ (expression (6)). Finally, we replace the selection condition C with C^b, apply C^b on the join results of D and U, and produce the final results by projecting the join results onto the attribute set $\mathcal{A}$ (expression (7)).

The cohort aggregation operator $\gamma^c_{\mathcal{L},e,f_A}$ is evaluated in following four steps.

$$S \leftarrow \pi_{\mathcal{A},A_t^b,D.A_t-A_t^b\rightarrow A_g}(R^e \underset{R^e.A_u=D.A_u}{\bowtie} D) \tag{8}$$

$$T \leftarrow \gamma_{\mathcal{L},\text{count}()\rightarrow\text{CohortSize}}\sigma_{A_t=A_t^b}(S) \tag{9}$$

$$U \leftarrow \gamma_{\mathcal{L},A_g,f(A)\rightarrow A_m}\sigma_{A_g>0}(S) \tag{10}$$

$$\gamma^c_{\mathcal{L},e,f_A}(D) \leftarrow T \underset{T.\mathcal{L}=U.\mathcal{L}}{\bowtie} U \tag{11}$$

First, we derive a temporary table S by joining D with R^e on A_u and calculating the age A_g (expression (8)). Second, we calculate the cohort size by first retrieving all birth activity tuples and then applying count() aggregation function on the cohort attribute set $\mathcal{L}$. The temporary results are stored in T (expression (9)). Third, we calculate the age metric by performing f_A on S based on the group by attribute set $(\mathcal{L}, A_g)$ and produce the results as U (expression (10)). Finally, we join T and U to get the final result (expression (11)).

With the implementation of cohort operators, a cohort query Q can be evaluated in two steps. First, we translate each cohort operator in Q into a view by `CREATE VIEW` statements based on the above translation schemes. Then, we use a `SELECT` statement to combine these views and feed this statement into the database for execution. The SQL based approach is easy to implement. However, the evaluation scheme is inefficient as it involves multiple joins.

3.2 Materialized View based Approach

A major bottleneck of SQL based approach presented in Section 3.1 is to find birth activity tuples, which requires to find birth time table R^e for the given birth action e and join

R^e with the activity table D on $D.A_u = R^e.A_u$ and $D.A_e = R^e.A_t^b$. To eliminate this cost, we can adopt a materialized view approach which materializes birth activity tuples in the original activity table for a particular birth action. Once the birth activity tuples are materialized, cohort operations can be implemented with fewer joins.

Formally, given a birth action e and a birth attribute set $\mathcal{A}^b = \{A_i, \ldots, A_{i+k}\}$ where $k \leq n-3$ and for each $A_j \in \mathcal{A}^b$, $A_j \in \mathcal{A} \wedge A_j \notin \{A_u, A_t, A_e\}$, we create a materialized view V as

$$T \leftarrow \pi_{A_u, A_t^b, \mathcal{A}^b} \sigma_{A_t = A_t^b} (R^e \underset{R^e.A_u = D.A_u}{\bowtie} D) \tag{12}$$

$$V \leftarrow \pi_{\mathcal{A}, \mathcal{A}^b, A_t^b, (A_t - A_t^b) \to A_g} (T \underset{T.A_u = D.A_u}{\bowtie} D) \tag{13}$$

The materialized view V is built in two steps. In the first step, we first join R^e with D on A_u and retrieve all birth activity tuples. Then, the results are projected on $(A_u, A_t^b, \mathcal{A}^b)$ and stored in T (expression (12)). In the second step, we join T with D on A_u and project the join results onto the desired attribute set; We also add the age (A_g) attribute in the materialized view V to avoid its computation during the evaluation of cohort aggregation operators (expression (13)).

Consider the activity table in Table 1, suppose we choose the launch action as the birth action and $\mathcal{L}^b = \{\text{Country}\}$ as the birth attribute set. Table 5 shows the corresponding materialized view. In Table 5, the attribute A_t^b is named BirthTime, the birth attribute Country is named BirthCountry and the attribute A_g is named Age.

With V, the cohort operators can be evaluated with fewer joins. The birth selection operator $\sigma^b_{C,e}$ is now evaluated as

$$\sigma^b_{C,e}(D) \leftarrow \sigma_{C^b}(V)$$

where C^b is the rewritten condition of C by replacing each attribute A in C with its corresponding birth attribute $\mathcal{A}^b$ in V. For example, the $\sigma^b_{\text{Country=China,launch}}$ operation on Table 5 is evaluated as a standard selection operation $\sigma_{\text{BirthCountry=China}}$.

The age selection operator $\sigma^g_{C,e}$ is implemented as

$$\sigma^g_{C,e}(D) \leftarrow \pi_{\mathcal{A}} \sigma_{C^b \vee A_t = A_t^b}(V)$$

where C^b is the rewritten condition of C by replacing each `Birth(A)` function call with the corresponding birth attribute A^b in $\mathcal{A}^b$.

Finally, the cohort aggregation operator $\gamma_{\mathcal{L},e,f_A}$ is evaluated by simply removing expression (8) in the implementation presented in Section 3.1 and replacing S with V in the rest expressions.

$$T \leftarrow \gamma_{L,\text{count()} \to \text{CohortSize}} \pi_{A_u}(V)$$
$$U \leftarrow \gamma_{L,A_g,f(A)}(V)$$
$$\gamma_{L,e,f_A}(D) \leftarrow T \underset{T.L=U.L}{\bowtie} U$$

The materialized view V that we introduced above improves the query performance by eliminating multiple joins. Furthermore, a standard B-tree index can also be built on birth attributes in V for speeding up birth selection operations. Unfortunately, for age selection operation, no index proposed so far can be used for efficient retrieval of qualified activity tuples, as according to Definition 4, the birth tuple should always be selected no matter if it passes the selection condition.

However, the materialized view technique has two major limitations. First, to employ V for query processing, every possible attribute A that appears in the condition C of the birth selection operator $\sigma^b_{C,e}$ or in the `Birth(A)` function of the age selection operator $\sigma^g_{C,e}$ must have its corresponding birth attribute A^b in V. Second, the materialized view V can only be used for processing cohort queries whose birth action e in the query is the same birth action that is used for the creation of V.

To solve the first issue, we can include every non-primary key attribute $A \in \mathcal{A}$ in the birth attribute set $\mathcal{A}^b$. To solve the second issue, we can create a set of materialized views $\mathcal{V} = \{V_1, \ldots, V_n\}$ where each materialized view $V_i \in \mathcal{V}$ is created for a specific birth action $e_i \in \text{Dom}(A_e)$. Unfortunately, both of the aforementioned approaches significantly increase the storage space and maintenance costs of materialized view management. Suppose our cohort query workload consists of n birth actions and queries m birth attributes. For each birth action, we need to perform 2 joins (expression (12) and (13)) and add to the original activity table $m + 2$ additional columns, m of which are birth attributes and the other two of which are the birth time and age attribute, respectively. As a result, creating a materialized view for each birth action requires $2n$ joins and generates $(m + 2) \times n$ additional columns in total. Obviously, even for modest setting, e.g., $n = 20$ and $m = 20$, the total number of joins and new attributes added is quite large. Therefore, this technique is only useful in scenarios where the cohort queries involve a very few number of birth actions and birth attributes.

3.3 Cohort Algorithms

The major problem of the materialized view approach presented in Section 3.2 is scalability, and this section develops a set of cohort algorithms to conquer the scalability issue. A key observation is that all cohort operators need to process birth activity tuples. Therefore, searching birth activity tuples constitutes the critical execution path in cohort query processing, and we can thus speed up the cohort query processing by optimizing the birth activity tuples searching operation. The optimization is performed in two steps. First, we sort the activity table D according to (A_u, A_t, A_e) during the load phase. The sorted activity table has two nice properties: 1) activity tuples of the same user are clustered together, called user clustering property; 2) For each user i, the activity tuples of i are stored in increasing order of time, called time ordering property. With these two properties, we can efficiently find the birth activity tuple for any birth action e in a single sequential scan. Suppose the activity tuples of user i is stored between d_j and d_k (the user clustering property). For any birth action e, we iterate each tuple between d_j and d_k to locate the birth activity tuple of i, and the first tuple d_b satisfying $d_b[A_e] = e$ is the desired birth activity tuple $d^{i,e}$. In the second step, we develop a set of sort-aware cohort operators which utilize the two properties of the sorted activity table for efficient cohort operations [5]. The sort-aware cohort algorithms will be refined in Section 4 and implemented in our COHANA query engine.

We now present the implementation of $\sigma^b_{C,e}$ and $\gamma^c_{\mathcal{L},e,f_A}$

[5]We shall note that even though the activity table is sorted, the existing relational operators still cannot perform efficient cohort operation since they are not coded to utilize such sorted storage for efficient birth tuple location.

Table 5: Materialized View of Table 1

Player	Time	Action	Role	Country	Gold	BirthTime	BirthCountry	Age
001	2013/05/19:1000	launch	dwarf	Australia	0	2013/05/19:1000	Australia	0
001	2013/05/20:0800	shop	dwarf	Australia	50	2013/05/19:1000	Australia	1
001	2013/05/20:1400	shop	dwarf	Australia	100	2013/05/19:1000	Australia	1
001	2013/05/21:1400	shop	assassin	Australia	50	2013/05/19:1000	Australia	2
001	2013/05/22:0900	fight	assassin	Australia	0	2013/05/19:1000	Australia	3
002	2013/05/20:0900	launch	wizard	United States	0	2013/05/20:0900	United States	0
002	2013/05/21:1500	shop	wizard	United States	30	2013/05/20:0900	United States	1
002	2013/05/22:1700	shop	wizard	United States	40	2013/05/20:0900	United States	2
003	2013/05/20:1000	launch	bandit	China	0	2013/05/20:1000	China	0
003	2013/05/21:1000	fight	bandit	China	0	2013/05/20:1000	China	1

Algorithm 1: $\sigma^b_{C,e}(D)$ operator implementation

```
   Input : A sorted activity table D and a birth action e
1  GetBirthTuple(d, e)
2      i ← d[A_u]
3      while d[A_u] = i ∧ d[A_e] ≠ e do
4          d ← D.GetNext()
5      return d

6  Open()
7      D.Open()
8      u_c ← ∅
9      Found ← false

10 GetNext()
11     d ← D.GetNext()
12     if d[A_u] ≠ u_c then
13         d^b ← GetBirthTuple(d, e)
14         u_c ← d^b[A_u]
15         Found ← C(d^b)
16     return Found = true ? d : ∅
```

Algorithm 2: $\gamma_{\mathcal{L},e,f_A}(D)$ operator implementation

```
   Input : A sorted activity table D, a birth action e, a cohort
           attribute list L
1  Open()
2      D.Open()
3      H^c ← ∅             // Cohort Size hash table
4      H^g ← ∅            // Cohort metric hash table
5      d^b ← ∅
6      u_c ← ∅
7      while D is not exhausted do
8          d ← D.GetNext()
9          if u_c ≠ d[A_u] then
10             d^b ← D.GetBirthTuple(d, e)
11             u_c ← d^b[A_u]
12             H^c[d^b[L]] + +
13         else
14             g ← d[A_t] − d^b[A_t]
15             update H^g[d^b[L]][g] with f_A(d)

16 GetNext()
17     Retrieve next key (c, g) from H^g
18     return (c, g, H^c[c], H^g[c][g])
```

operators in Algorithm 1 , Algorithm 3 and Algorithm 2 respectively. Each cohort operation is implemented as a standard operator, which is a group of three functions: `Open()` for initializing the operator, `GetNext()` for returning the next tuple in the result set and `Close()` for terminating the iteration. We omit the `Close()` implementation since the implementation is trivial: we just close the input activity table by calling D.`Close()`.

Algorithm 1 presents the implementation of $\sigma^b_{C,e}$ operator. The algorithm employs an auxiliary function `GetBirthTuple`(d, e) (line 1 – line 5) for finding the birth tuple of user $i = d[A_u]$ given that d is the first activity tuple of i in the underlying storage and e is a birth action. The `GetBirthTuple()` function finds i's birth tuple by iterating each next tuple $d \in D$ and checks whether d belongs to i and whether the value $d[A_e]$ is the birth action e (line 3). The first tuple d matching the condition is returned as the birth tuple.

To evaluate $\sigma^b_{C,e}$, Algorithm 1 first opens the input activity table D and initializes two global variables (line 8 – line 9): u_c which points to the current processing user and Found, a boolean flag which indicates whether the current processing tuple should be returned as a result. In the `GetNext()` function, we first retrieve the next tuple d from the input (line 11). Then, we check whether the user of d, i.e., $d[A_u]$, is the current processing user (line 12). If the check fails, we look for a new user and then call `GetBirthTuple`(d, e) to get the birth tuple d^b of $d[A_u]$ (line 13). Thereafter, we apply the selection condition on d^b and update u_c and the Found flag accordingly (line 14 – line 15). We finally return d if Found is true (line 16).

Algorithm 3 presents the implementation of $\sigma^g_{C,e}$. The `Open()` function opens the input activity table D and initializes three global variables: u_c which stores the current processing user, d^b which holds u_c's birth tuple and the boolean indication flag Found (line 2 – line 5). The `GetNext()` function first retrieves the next tuple d from D (line 7). Then, it checks whether $d[A_u]$ is the current processing user u_c. If the check fails, a new user is found. We update u_c and d^b and return the birth tuple of newly found user (line 9 – line 12). If the check succeeds, d must be an age tuple of u_c. We rewrite the condition C by replacing each occurrence of `Birth(`A`)` with the value $d^b[A]$ and apply it to d (line 14 – line 15). Finally, the activity tuple d is returned if Found is true.

Algorithm 2 presents the implementation of $\gamma^c_{\mathcal{L},e,f_A}$ operator. The `Open()` function implements the main logic of $\gamma^c_{\mathcal{L},e,f_A}$. The function first initializes its data structures: a hash table H^c which stores the cohort size for each cohort; a hash table H^g which stores aggregation result of f_A for each cohort; u_c, the current processing user and d^b, the birth tuple of u_c (line 2 – line 6). Then, the `Open()` function iterates through each activity tuple $d \in D$ in a while loop (line 7). For each d, we first check whether $d[A_u]$ is the current processing user u_c (line 9). If $d[A_u]$ is a new user, we update d^b and u_c and then increment the cohort size $H^c[d_\mathcal{L}]$

Algorithm 3: $\sigma^{g}_{C,e}(D)$ operator implementation

Input: A sorted activity table D and a birth action e

`Open()`
    D.`Open()`
    $d^b \leftarrow \emptyset$
    $u_c \leftarrow \emptyset$
    Found $\leftarrow$ `false`

`GetNext()`
    $d \leftarrow D$.`GetNext()`
    if $d[A_u] \neq u_c$ **then**
        $d^b \leftarrow D$.`GetBirthTuple`(d, e)
        $u_c \leftarrow d^b[A_u]$
        $d \leftarrow d^b$
        Found $\leftarrow$ `true`
    else
        $C' \leftarrow$ replace `Birth`(A) with $d^b[A]$ in C
        Found $\leftarrow C'(d)$
    return Found $=$ `true` ? d : $\emptyset$

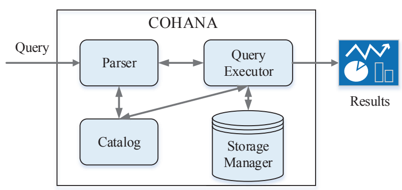

Figure 3: COHANA Architecture

(line 10 – line 12). If $d[A_u]$ is not a new user, we calculate the age of $d[A_u]$, apply the aggregate function f_A and update H^g accordingly (line 14 – line 15). The `GetNext()` function simply retrieves each entry from H^g and returns $(c, g, H^c[c], H^g[c][g])$ as a result (line 17 – line 18). The runtime performance of Algorithm 1 – Algorithm 2 is bound by $\Theta(|D|)$.

3.3.1 Discussions on Cohort Algorithms

We now analyze the runtime performance of Algorithm 1- Algorithm 2. For all three algorithms, it can be clearly seen that to produce the result set, each activity tuple in the input activity table D is examined only once. Hence the run time performance of the three algorithms grows linearly with the number of activity tuples in the input and therefore is bound by $\Theta(|D|)$. Our cohort algorithms require that the input activity table to be sorted. At first glance, maintaining a sorted table may be costly for tuples insertion. However, as shown in C-store [15], this problem can be solved by maintaining a writable store in memory and a read-only sorted store on disk and using a tuple mover to periodically merge these two stores.

4. THE COHANA SYSTEM

This section refines the basic cohort algorithms presented in Section 3.3 and introduce more optimizations for efficient cohort query processing. We developed COHANA, a query engine for cohort analytics, to demonstrate how these techniques can be readily integrated into a real system. COHANA is built from scratch in Java. We shall present the centralized version of COHANA in this paper, and will present its distributed version in the next paper.

As shown in Figure 3, COHANA shares the similar architecture with other read-mostly analytical database systems and consists of three major modules: a catalog module for managing the schema of activity tables, a storage module for persisting activity tables, and a query processing module for processing cohort queries. Efficient cohort query processing is achieved by a fine-tuned compressed columnar storage for sorted activity tables, applying sort-aware cohort operators for performing cohort operations, pushing down birth selection operator, skipping unqualified users for query optimization, and finally a new `UserCount()` aggregation function for fast calculation of cohort size.

4.1 Schema Management in COHANA

The catalog module of COHANA is responsible for managing the schema information of activity tables. As described in Section 2, an activity table is a special relational table with at least three attributes A_u, A_t and A_e. To declare the schema of an activity table, we add three attribute types: `UserKey`, `Time`, `Action` to the standard SQL `CREATE TABLE` statement. If an attribute is declared as a type of `UserKey`, `ActionTime` or `Action`, it will be served as the A_u, A_t or A_e attribute respectively. Attributes without type annotations are normal relational attributes. The schema of Table 1 is declared as follows

```
CREATE TABLE GameActions (
    player string UserKey,
    time timestamp ActionTime,
    action string Action,
    role string,
    country string,
    Gold int
)
```

In addition to persisting the schema information, the catalog module also checks the validity of the activity table schema, namely there is one and only one `UserKey`, `ActionTime` and `Action` attribute declared. The catalog module also automatically adds the primary key constraint before committing the schema information.

4.2 Storage Module

As we have mentioned, the data model for cohort analysis is a relational activity table with the primary key constraint on (A_u, A_t, A_e), i.e., the user, timestamp and action attributes. To support efficient cohort analysis, multiple optimization techniques are employed for the storage of the activity table. Briefly speaking, we employ a two-level storage format where the activity table is first horizontally and evenly partitioned into multiple data chunks such that the activity tuples of each users are included in only one chunk (this is trivial as the activity table is sorted with respect to (A_u, A_t, A_e)), and each data chunk is then vertically partitioned and persisted column by column.

We employ various compression techniques for the storage of columns in each data chunk. To this end, we first build a global index for each column. For a string column, the global index is an sorted array of the unique values, and for an integer column[6], the global index just contains the MIN

[6] A numeric column can always be transformed into an inte-

and MAX values. The global indices of all columns form a special meta-chunk, and is persisted in front of data chunks.

The global indices are then used to compress the columns of each data chunk. We mainly consider three encoding schemes for compression: Run-Length-Encoding (RLE), dictionary encoding and delta encoding. The user column A_u is always compressed by RLE and represented as a sequence of triples (u, f, n), where u is the user in A_u, f is the position of the first appearance of u in the column, and n is the number of appearances of u in the column. Integer columns, e.g., measure columns or timestamp columns, are compressed by delta encoding. For an integer column A, we retrieve the MIN value from A's global index and subtract it from the values in A so that the resultant delta values are within a smaller range and hence can be represented with less number of bits. The MIN and MAX values of the delta values are also stored preceding the delta values as a range index to prune the residing chunk during the processing of queries which fall outside the range. String columns such as A_e are hierarchically represented. First, we associate each value in a string column A with its global-id, which is the position of this value in the global index of A, and store in ascending order the distinct global-ids into an array called a chunk index. Second, for each value in A, we compute its chunk-id, which is the position in the chunk index at which the respective global-id locates. The chunk-ids of all values in A are then stored immediately after the chunk index in the same order as the values appear in A. This two-level dictionary encoding scheme is identical to the one introduced in [10] where more details of this encoding scheme can be found.

As can be inferred, many columns as well as the chunk indices are represented as integer arrays. We therefore further employ integer compression techniques to reduce the storage space. For each integer array, we compute the minimum number of bits, denoted by n, to represent the maximum value of the array, and then sequentially pack as many values as possible into a 64-bit integer such that each value only occupies exactly n bits of the 64-bit integer. An advantage of this compression manner is that each original value can be directly read from the certain n bits of the respective 64-bit integer, hence eliminating the cost of decompression.

4.3 Query Evaluation

COHANA provides a SQL `SELECT` like statement for specifying cohort queries. The main construct of a cohort `SELECT` statement consists of `BIRTH FROM`, `AGE ACTIVITIES IN` and `COHORT BY` which respectively corresponds to the $\sigma^b_{C,e}$, $\sigma^g_{C,e}$ and $\gamma^c_{\mathcal{L},e,f_A}$ operations. Two keywords `CohortSize` and `Age` are also added for data analysts to retrieve values from the calculated attributes in the results of the $\gamma^c_{\mathcal{L},e,f_A}$ operation. The query in example 1 is specified as follows. Note that the action in the `BIRTH FROM` clause specifies the birth action for the whole query.

```
SELECT country, CohortSize, Age,
        sum(Gold) as TotalGold
FROM GameActions
BIRTH FROM action = "launch" AND
    role = "dwarf"
AGE ACTIVITIES IN action = "shop"
COHORT BY country
```

ger column by multiplying each value in the column by an appropriate scale.

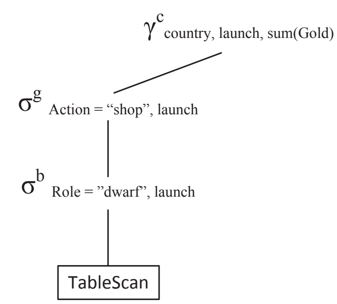

Figure 4: Query Plan for Example 1

Overall, to evaluate a cohort query such as the one above, a query plan, i.e., a tree of physical operators, is first produced. Then, we optimize this query plan by pushing down birth selection operator, and execute the refined query plan against each data chunk. The final results are produced by merging all partial aggregation results obtained for each data chunk. The query evaluation scheme is similar to other databases, and hence we only focus on the part which is unique to cohort query processing.

We implement four physical operators: `TableScan`, birth selection $\sigma^b_{C,e}$, age selection $\sigma^g_{C,e}$ and cohort aggregation $\gamma^c_{\mathcal{L},e,f_A}$. The projection operation is implemented in a preprocessing step: we collect all required columns at query parsing stage and then pass those columns to the `TableScan` operator which retrieves the values for each column.

The `TableScan` operator is specially designed for activity tables. Since the A_u column is always compressed by RLE, in addition to the `GetNext()` function which returns the next activity tuple from D, we add two new functions `GetNextUser()` and `SkipCurUser()` to accelerate the locating of the qualified activity tuples leveraging the property of RLE. In particular, the first function returns an RLE triple (u, f, n), where f and n respectively represents the offset of the first activity tuple and the number of activity tuples of the next user u; the `SkipCurUser()` function sets the internal file reader pointer of each column in the `TableScan` operator to $f+n$, thereby skipping the processing of the rest activity tuples of u. With these two functions, the activity tuples of unqualified users can be quickly skipped.

4.3.1 Query Optimization

For the moment, a cohort query only involves unary operations. In a query plan produced by COHANA, the root and the only leaf node are the aggregation operator, $\gamma^c_{\mathcal{L},e,f_A}$, and the *TableScan* operator, respectively, and between them is a sequence of the birth selection operators and the age selection operators. We push down the birth selection operators along the query plan so that they are always below the age selection operators. This push-down optimization is always feasible, since according to equation (1), we can swap the order of $\sigma^b_{C,e}$ and $\sigma^g_{C,e}$ operators in any sequence consisting of these two operators. Figure 4 shows the query plan for the cohort query of example 1.

The reason we employ this push-down optimization is that the sort-aware birth selection operator presented in Section 3.3 can be refined to efficiently skip the activity tuples of unqualified users by utilizing the two aforementioned auxiliary functions, i.e., `GetNextUser()` and `SkipCurUser()`, during the evaluation of $\sigma^b_{C,e}$. In the `GetNext()` function, the refined $\sigma^b_{C,e}$ operator calls D.`GetNextUser()` to retrieve the next triple (u, f, n) for a new user, and then calls `GetBirthTuple()` to retrieve the birth activity tuple $d^{u,e}$ of u. If the birth activity tuple satisfies the condition, it proceeds in the same way as Algorithm 1. Otherwise, it calls D.`SkipCurUser()` to skip the rest of the activity tuples of the current user. As a result, the refined $\sigma^b_{C,e}$ significantly reduces the number of activity tuples to be processed.

We shall emphasize two additional advantages of our approach. First, we always directly process compressed A_u column and therefore avoid the cost of decompression. Second, even though we use `SkipCurUser()` to skip the activity tuples of unqualified users, we actually do not pay the cost for random seek since we process all users sequentially.

As a final optimization for $\sigma^b_{C,e}$, we refine the implementation of `GetBirthTuple()` to cache the locations of birth activity tuples. The refined `GetBirthTuple()` holds a set of in-memory hash tables, each allocated for a specific birth action e and initialized as an empty table. Each time `GetBirthTuple`(d, e) is called, we first check whether the user $d[A_u]$ can be found in the hash table H allocated for e. If the check returns true, the `GetBirthTuple()` function directly reads the birth activity tuple from the location cached in H. Otherwise, it performs a sequential iteration over the activity tuples of the given user to find the birth activity tuple, and update H accordingly. This optimization further reduces the time spent in searching birth activity tuples.

4.3.2 Evaluation of $\gamma^c_{\mathcal{L},e,f_A}$

We incorporate the idea from [10] to improve the performance of Algorithm 2 by using a two dimension array in the inner loop for calculating aggregates.

Consider the cohort query in Example 1. To aggregate the qualified age activity tuples in a data chunk, we first identify the total number of unique values in the cohort column, i.e., Country, by checking the size of the chunk index of the Country column, denoted by n. Then, we retrieve the MIN and MAX value of Time column from its chunk index. The difference between MAX and MIN represents the maximum age of users in the data chunk.

Let $m = \text{MAX} - \text{MIN}$, we initialize a two dimensional array $A[n][m+1]$ to hold the aggregation results for each data chunk where $A[c][0]$ holds the size of cohort c and $A[c][g]$ holds the sum of Gold of cohort c at age g. The aggregation in Algorithm 2 is finally performed by replacing the two hash tables with the array $A[n][m+1]$ in the inner while loop. As shown in [10], the advantage of using an array in the inner loop of aggregation is that modern CPUs can highly pipeline array operations and thus significantly improve the performance of aggregation.

4.3.3 Optimization for User Count Calculation

One popular application of cohort analysis is to show the trend in user retention [1]. These cohort queries involve computing the distinct number of users at different ages. This computation is very costly in terms of memory and performance for fields with a large number of distinct values, such as A_u.

In COHANA, however, counting the number of distinct values of the A_u column is very simple. The chunk storage format that we employ ensures that the activity tuples of any user are included in only one chunk. Therefore, for each cohort at each age, we can simply perform the counting on each chunk and return the sum of the obtained numbers as the final result. We thus implement a `UserCount()` aggregation function which utilizes this property and a bitmap algorithm for the efficient counting of distinct users.

5. EXPERIMENTS

This section presents the performance study of cohort query processing. We mainly perform two sets of experiments. First, we compare the performance of different query evaluation schemes. We implement the SQL based approach and the materialized view based approach on two relational databases: MySQL and MonetDB, and compare the performance of these two systems with COHANA. In the second set of experiments, we focus on the performance study of COHANA and evaluate the effect of each query optimization technique that we propose for cohort query processing.

5.1 Experiment Settings

All experiments run on a high-end workstation. The workstation is equipped with a quad-core Intel Xeon E3-1270 v3 3.50GHz processor and 16GB of memory. The disk speed reported by `hdparm` is 15GB/s for cached reads and 150MB/s for buffered reads.

The dataset we used is produced by a real mobile game application. The dataset consists of 30M activity tuples contributed by 57,077 users worldwide from 2013-5-19 to 2013-06-26. In addition to the required user, action and action time attributes, we also include the country, city and role as dimensions and session length and gold as measures. Totally, users in the game played 16 actions. We choose the launch and shop actions as the birth actions.

5.2 Benchmark Settings

For MySQL, we choose MyISAM as our storage engine. For MonetDB, we accept all default settings without further tuning. For the SQL based approach, we manually translate the cohort query into a sequence of SQL commands for execution. For the materialized view based approach, we manually materialize the view using `CREATE TABLE AS` command. We materialize the birth time attribute, age attribute and a birth attribute set consisting of role, country and city attribute for each birth action (launch and shop). Totally, this materialization scheme adds ten additional columns to the original table by performing four joins. As a final tuning for the storage, we build an index on the birth time and each birth attribute column to speedup birth selection operations. For COHANA, we choose the chunk size to be 16K.

5.3 Benchmark Queries

We design four queries (described with COHANA's cohort query syntax) for the benchmark by incrementally adding the cohort operators we proposed in this paper. The first query Q1 evaluates a single cohort aggregation operator. The second query Q2 evaluates a combination of birth selection and cohort aggregation. The third query Q3 evaluates a combination of age selection and cohort aggregation. The fourth query Q4 evaluates a combination of all three cohort operators. For each query, we report the average execution time of six runs for each system.

Q1: For each country launch cohort, report the number of retained users who did at least one action since they first launched the game.

```
SELECT country, CohortSize, Age, UserCount()
FROM GameActions
BIRTH FROM action = "launch"
COHORT BY country
```

Q2: For each country launch cohort born in a specific date range, report the number of retained users who did at least one action since they first launched the game.

```
SELECT country, CohortSize, Age, UserCount()
FROM GameActions
BIRTH FROM action = "launch" AND
    time BETWEEN "2013-05-21" AND "2013-05-27"
COHORT BY country
```

Q3: For each country shop cohort, report the average gold they spent in shopping since they made first shop in the game.

```
SELECT country, CohortSize, Age, avg(gold)
FROM GameActions
BIRTH FROM action = "shop"
AGE ACTIVITIES IN action = "shop"
COHORT BY country
```

Q4: For each country shop cohort, report the average gold they spent in shopping in their birth country where they were born with respect to the dwarf role in a given date range.

```
SELECT country, CohortSize, Age, avg(gold)
FROM GameActions
BIRTH FROM action = "shop" AND
    time BETWEEN "2013-05-21" AND "2013-05-27" AND
    role = "dwarf" AND
    country IN ["China", "Australia", "United States"]
AGE ACTIVITIES IN action = "shop" AND
    country = Birth(country)
COHORT BY country
```

In order to investigate the impact of the birth selection operator and the age selection operator on the query performance of COHANA, we further design two variants of Q1 and Q3 by adding to them a birth selection condition (resulting in Q5 and Q6) or an age selection condition (resulting in Q7 and Q8). The details of Q5-Q8 are show below.

Q5: For each country launch cohort, report the number of retained users who did at least one action during the date range [d1; d2] since they first launched the game.

```
SELECT country, CohortSize, Age, UserCount()
FROM GameActions
BIRTH FROM action = "launch" AND
    time BETWEEN d1 AND d2
COHORT BY country
```

Q6: For each country shop cohort, report the average gold they spent in shopping during the date range [d1; d2] since they made first shop in the game.

```
SELECT country, CohortSize, Age, avg(gold)
FROM GameActions
BIRTH FROM action = "shop" AND
    time BETWEEN d1 AND d2
AGE ACTIVITIES IN action = "shop"
COHORT BY country
```

Table 6: Results for Storage Space

Storage Format	Disk Space (GB)
Raw	3.63
MySQL-Raw	3.41
MySQL-MV	6.14
Monet-Raw	0.94
Monet-MV	1.80
COHANA	0.31

Table 7: Query Performance Results in Seconds

Query	My-S	My-M	Mon-S	Mon-M	COHANA
Q1	3.9hr	1062.12	5.10	1.88	0.27
Q2	1.6hr	14.80	0.67	0.26	0.02
Q3	2.1hr	156.49	2.55	1.29	0.32
Q4	1.1hr	5.47	0.58	0.20	0.01

Q7: For each country launch cohort at each less than g age, report the number of retained users who did at least one action since they first launched the game.

```
SELECT country, CohortSize, Age, UserCount()
FROM GameActions
BIRTH FROM action = "launch" AND
AGE ACTIVITIES in Age < g
COHORT BY country
```

Q8: For each country shop cohort at each less than g age, report the average gold they spent in shopping since they made first shop in the game.

```
SELECT country, CohortSize, Age, avg(gold)
FROM GameActions
BIRTH FROM action = "shop" AND
AGE ACTIVITIES IN action = "shop" AND
    Age < g
COHORT BY country
```

5.4 Benchmark Results

Table 6 shows the space of raw data and the storage budget each system spends for data import. For MySQL and MonetDB, we report the storage space that the two systems used for storing the raw activity table and storing the materialized view.

As can be seen from Table 6, the raw data dumped in CSV format from the game application occupies 3.63GB disk space. MySQL requires 3.41GB and 6.14GB for storing the raw activity table and the materialized view respectively. MonetDB adopts a more compact storage format and only spends 0.94GB and 1.80GB space budget for storing the raw activity table and the materialized view. With the combination of RLE, delta and two-level dictionary compression scheme, COHANA only uses 0.31GB storage space for persisting the raw activity table, which means a compression ratio of 12X.

Table 7 reports the execution time that each system takes to execute the four queries. As expected, the SQL based approach is the slowest approach as it needs joins for processing cohort queries. On the raw table format, MySQL spends more than an hour to process a cohort query (My-S), mainly because it employs the nested loop join algorithm for join processing. For large tables, the nested loop join technique becomes the bottleneck of the query processing. MySQL runs much faster on the materialized view storage format

(My-M) since, under this storage format, it does not require joins on source tables for query evaluation. However, for Q1, MySQL still takes more than 15mins to complete. This is because Q1 is a retention query which requires performing distinct count aggregation. The evaluation of the expensive distinct count aggregation therefore becomes the bottleneck of this query.

As a very fast columnar database, MonetDB is able to perform all four queries on both storage formats (raw and materialized view) within a reasonable time. However, as we have expected, performing cohort queries on raw format (Mon-S) is much slower (2X $\sim$ 3X) than processing cohort queries on materialized view (Mon-M) format.

COHANA is obviously much faster than the other two systems. Compared to MonetDB, for all queries but Q3, COHANA runs 6X - 20X and 18X - 58X faster than Mon-M and Mon-S, respectively. For MySQL, COHANA runs 500X - 4000X faster than My-M and three orders faster than My-S. The main reasons are: 1) a fine tuned compressed columnar storage format, 2)efficiently skipping unqualified users, 3) array based aggregation, and 4) an efficient implementation of `UserCount()`.

The reason for the small performance gap between COHANA and other systems on Q3 is that this query does not have a birth selection condition and employs the `avg()` function for aggregation. Therefore, COHANA cannot speedup this query using birth selection optimization and the optimized `UserCount()` aggregate function. However, with the benefit from the array based aggregation and caching locations of birth activity tuples, COHANA stills runs 3.4X faster than Mon-M and 6.7X faster than Mon-S.

5.5 Performance Study of COHANA

The results in Table 6 and Table 7 show that, overall, the optimizations we made for COHANA are promising. However, we are still interested in the effect of each optimization technique we implemented in COHANA. This section conducts a set of experiments to study the performance of COHANA by evaluating the effect of each query optimization technique that we propose for cohort query processing.

5.5.1 *Effect of Chunk Size*

We first study the effect of chunk size on the storage space and query performance of COHANA. To this end, we vary the chunk size from 1K (1024 activity tuples per-chunk) to 1M and study the corresponding variance in the storage space and query performance of COHANA.

Figure 5 presents the storage space COHANA requires for the activity table compressed with different chunk size, and Figure 6 presents the query performance of COHANA under different chunk size. It is clearly seen that, in Figure 5, increasing the chunk size also augments storage cost. This is because that an increase in the size of a chunk will lead to more players included in that chunk. As a result, the number of distinct values in the columns of each chunk also increases, which in turn requires more bits for encoding values. On the other hand, although a smaller chunk size results in a better compression ratio, it increases the overhead for processing the chunk header and loading chunk index. Consequently, we see a slight increase in query time as shown in Figure 6. There exists another negative effect of large chunk size that can be found in Figure 6. Specifically, a large chunk size will introduce more cache misses in chunk level aggregation. As mentioned before, COHANA adopts a two-dimensional array based algorithm for aggregating age activity tuples in data chunks. This algorithm works best if the entire array fits into L2 cache. As the chunk size increases, there are more and more users in the data chunk,. Hence, the total number of cohort in each chunk increases, resulting in a large array which may not be fit into L2 cache. If this happens, the query performance is affected (as demonstrated by the slight increase in the query execution time of Q1, Q2 and Q3 in Figure 6). Overall, the optimal chunk size is collectively determined by data distribution and query workload. We leave the study of optimal chunk size as a future work.

Based on the results of this experiment, we choose the chunk size to be 16K for all the rest of experiments.

5.5.2 *Effect of Birth Selection*

In Section 4.3.1, we claim that the running time of COHANA is bound by $O(n)$ where n is the total number of qualified users. This experiment studies the query performance of COHANA with respect to the birth selection selectivity. We run Q5 and Q6, which are an variant of Q1 and Q3, respectively. The date range $[d_1, d_2]$ of Q5 and Q6 is chosen such that the qualified players account for 10% to 100% of the whole population.

Figure 7 presents the results of this experiment, which clearly shows that the performance of COHANA grows linearly with the number of qualified users. We attribute this expected linear growing trend to the optimization of pushing down the birth selection operator and the refined birth selection algorithm which is capable to skip unqualified users.

5.5.3 *Effect of Caching Birth Location*

To improve the performance of fetching birth activity tuples, COHANA employs additional memory to cache the locations of birth activity tuples for each birth action at query time. This experiment studies the effect of birth location caching. Since the launch action is the first action of all players, caching the location of the launch birth action does not make sense as it cannot reduce the time to locate the launch birth tuple. Thus, we again choose Q6 for this experiment. We run Q6 with varying birth selection selectivity and compare the query performance between the cases with/without birth locations caching enabled.

Figure 8 presents the results for this experiment. Overall, enabling birth location caching improves the query performance by a factor of 10%, and this performance improvement is consistent over all selectivity.

5.5.4 *Effect of Age Selection*

The main work of a cohort query is aggregating age activity tuples. Therefore, this experiment studies the query performance of COHANA under different age selection conditions. To this end, We run Q7 and Q8 by varying g from 1 day to 14 days. [7].

Figure 9 presents the results of this experiment. It can be seen from this figure that the processing times of Q7 and Q8 exhibit different trend. Specifically, the processing time of Q7 increases almost linearly, while for Q8, the processing time increases slowly. The reason for this difference is that the performance of Q7 is bounded by the number of distinct

[7]We observed that the total number of actions players produced per-day is relatively stable in their first 14 days and drops dramatically afterwards – the aging effect

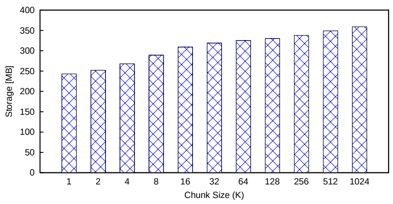

Figure 5: Storage Space under Different Chunk Size

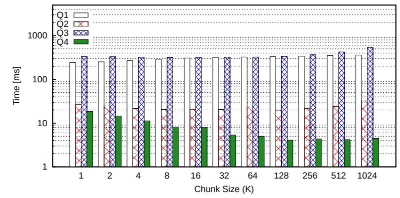

Figure 6: Query Performance under Different Chunk Size

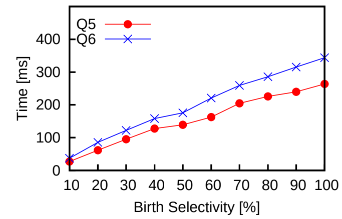

Figure 7: Effect of Birth Selection

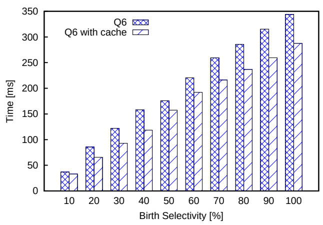

Figure 8: Effect of Birth Selection Cache

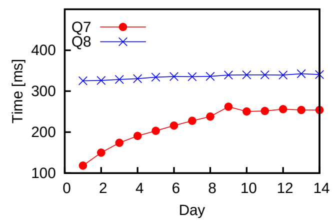

Figure 9: Effect of Age Selection

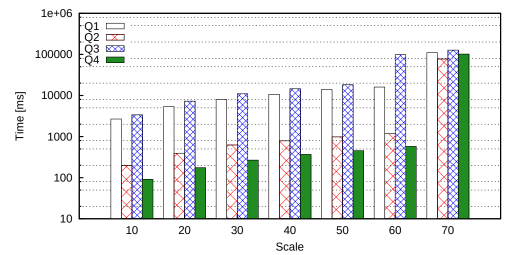

Figure 10: Dataset Scaling

users within the given age range, which grows almost linearly with age range. For Q8, the processing time mainly depends on the aggregation performed upon the shop age activity tuples, whose number grows slowly with the age – the aging effect.

5.5.5 *Scalability Experiments*

With compression, COHANA reduces the size of raw dataset from 3.63GB to 309MB. Thus, the compressed dataset can be entirely memory-mapped into main memory. We are interested in the scalability of COHANA. Namely, how does the performance of COHANA vary with the size of the compressed activity table, especially in the case where the compressed table can not be accommodated into memory?

We treat the original 3.63GB dataset as scale 1 and scale the dataset by replication. The replication is performed by replacing the player IDs with new 57,077 player IDs and countries in the original dataset with new countries. As such, in scale 2, we have 7.26GB data composed of the original dataset and one replicated dataset. We vary the scale from 10 to 70 for the scalability experiment. For scale 50 upwards, the sizes of the raw dataset and the compressed activity table are at least 180GB and 15GB, respectively. Given that the total size of our memory is 16GB and only 12GB is free for memory mapping (the rest 4GB is reserved by OS), it is obviously impossible for OS to cache the entire compressed activity table in memory for scale 50 upwards, and in such cases, we expect the time of query processing to be gradually dominated by hard disk accesses.

Figure 10 presents the results of this experiment. It can be inferred from this figure that the execution time of Q1, Q2 and Q4 undergoes an almost linear increase when the scale goes from 10 to 60, and then a dramatic jump on scale 70. The similar trend also applies to Q3, with the exception that the linear increase in processing time terminates at scale 50, instead of scale 60. The reasons for the trend in query processing time are as follows. At first, when the scale increases from 10 to 60 (50 for Q3), the whole data needed for query processing can be entirely memory-mapped into the main memory, and the processing time hence increases linearly as a result of the linear increase in the number of processed activity tuples. Afterwards, the data required for query processing is no longer able to be completely memory-mapped, leading to disk accesses which are several orders of magnitude slower than the accesses of memory. Hence, the execution time of the four queries is gradually dominated by disk accesses, and increases much faster than before. As shown in Figure 10, the processing time of each query at scale 70 (60 for Q3) is one or two orders of magnitude higher than that at a smaller scale.

One may notice that our free memory is 12 GB in the experimental platform, and may expect the linear increase in the processing time to terminate at scale 40, at which the compressed activity table occupies all the available memory. But it does not happen. The actual execution time jump occurs at scale 50 and scale 60. This is because of the way the operating system performs memory mapping, which only loads the necessary data, i.e., data to be accessed, from the disk. For Q1, thanks to the columnar storage, we only need to read from the disk two columns (country and action), both of which have a limited cardinality (150 and 16, respectively) and hence can be compressed in a very compact manner. For Q2 and Q4, although more columns need to be accessed, many of the chunks that do not satisfy the predicates will be pruned and hence not loaded, leading to less memory usage. For Q3, since all chunks must be processed and the compression ratio of column gold is not as high as the other two (country and action), the tuning point of execution time thus comes earlier than that of the other three queries, as shown in Figure 10.

6. RELATED WORK

The work related to ours is the relational database support for data analysis and cohort analysis. The work in [9] introduces a `CUBE` operator to SQL and enables the so called OLAP data analysis. A typical OLAP query requires an aggregation over a large dataset, an operation which is not well supported by traditional row-oriented databases. Hence, columnar databases are built for solving the efficiency issue [7, 13, 15]. Techniques such as data compression [16, 18], query processing on compressed data [4, 6, 11], array based aggregation [5, 17], and materialized view approaches [14] are proposed for speeding up OLAP queries. Similar to OLAP, a cohort query also requires large aggregations. Therefore, we adopt many techniques from columnar databases for accelerating cohort queries, a columnar storage format similar to [10], directly processing RLE compressed columns and a two dimensional array based aggregation algorithm.

In social science literatures, the term cohort is referred to a number of individuals who experienced a particular event, e.g., birth, marriage, during a specified period of time [8], e.g., a day, week or year. For example, people who were born in 1977 form a 1977 birth cohort. The definition of cohort presented in this paper is more general than the definition used in social science. Similar to a social science's cohort, the cohort in this paper is also a group of people who were born with respect to a specific birth action. However, instead of solely using the time attribute associated with the birth action for identifying a cohort, we can choose any values in the attribute set (exclude A_u and A_e) of birth tuples to define a cohort. At of this writing, Google has released a beta version of cohort report plugin for its popular Web analytic service [3]. This cohort report tool, however, only supports cohorting people by acquisition date. MixPanel [1] and RJMetrics [2] offer more advanced and social science style cohort query facilities.

7. CONCLUSIONS

Cohort analysis is a general and powerful analysis tool for finding unusual activity trend in large activity tables. In this paper, we have investigated database support for cohort analysis, which is the first of its kind. We have introduced an extended relational data model for representing activity data and proposed several novel cohort operators to facilitate a cohort analysis task. We have also investigated three query evaluation schemes. Our experimental results show that our newly proposed cohort query evaluation scheme achieves significantly better performance than the other two schemes. This empirical evidence confirms the effectiveness of building a special purpose query engine for cohort query. For future work, we plan to consider binary operations on activity tables and optimization techniques for age selection.

8. REFERENCES

[1] Retention. https://mixpanel.com/retention/.

[2] Rjmetrics. https://rjmetrics.com/.

[3] Use the cohort analysis report. https://support.google.com/analytics/answer/6074676?hl=en.

[4] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[5] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *VLDB*, pages 506–521, 1996.

[6] S. Amer-Yahia and T. Johnson. Optimizing queries on compressed bitmaps. In *VLDB*, pages 329–338, 2000.

[7] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[8] N. D. Glenn. *Cohort Analysis*. Sage Publications, Inc., London, 2005.

[9] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, pages 152–159, 1996.

[10] A. Hall, O. Bachmann, R. Büssow, S. Ganceanu, and M. Nunkesser. Processing a trillion cells per mouse click. *PVLDB*, 5(11):1436–1446, 2012.

[11] Y. Li and J. M. Patel. Bitweaving: Fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.

[12] Y. Li and J. M. Patel. WideTable: An accelerator for analytical data processing. *PVLDB*, 7(10):907–918, 2014.

[13] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3):231–246, 2000.

[14] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In *VLDB*, pages 75–86, 1996.

[15] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[16] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[17] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *SIGMOD*, pages 159–170, 1997.

[18] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, pages 59–70, 2006.