DEX: Query Execution in a Delta-based Storage System

Amit Chavan
University of Maryland, College Park
amitc@cs.umd.edu

Amol Deshpande
University of Maryland, College Park
amol@cs.umd.edu

ABSTRACT

The increasing reliance on robust data-driven decision-making across many domains has made it necessary for data management systems to manage many thousands to millions of *versions* of datasets, acquired or constructed at various stages of analysis pipelines over time. *Delta encoding* is an effective and widely-used solution to compactly store a large number of datasets, that simultaneously exploits redundancies across them and keeps the average retrieval cost of reconstructing any dataset low. However, supporting any kind of rich retrieval or querying functionality, beyond single dataset checkout, is challenging in such storage engines. In this paper, we initiate a systematic study of this problem, and present DEX, a novel stand-alone delta-oriented execution engine, whose goal is to take advantage of the already computed deltas between the datasets for efficient query processing. In this work, we study how to execute *checkout*, *intersection*, *union* and *t-threshold* queries over record-based files; we show that processing of even these basic queries leads to many new and unexplored challenges and trade-offs. Starting from a query plan that confines query execution to a small set of deltas, we introduce new transformation rules based on the algebraic properties of the deltas, that allow us to explore the search space of alternative plans. For the case of checkout, we present a dynamic programming algorithm to efficiently select the optimal query plan under our cost model, while we design efficient heuristics to select effective plans that vastly outperform the base checkout-then-query approach for other queries. A key characteristic of our query execution methods is that the computational cost is primarily dependent on the size and the number of deltas in the expression (typically small), and not the input dataset versions (which can be very large). We have implemented DEX prototype on top of `git`, a widely used version control system. We present an extensive experimental evaluation on synthetic data with diverse characteristics, that shows that our methods perform exceedingly well compared to the baseline.

Keywords

Delta encoding; versioning; query processing

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3035918.3064056

1. INTRODUCTION

Data-driven methods and products are becoming increasingly common in a variety of communities, leading to a huge diversity of datasets being continuously generated, modified, and analyzed. Many more datasets are typically created as intermediate results of data analysis pipelines. An increasingly important consideration for the underlying data management systems is that, all of these datasets and their versions over time need to be stored and queried for a variety of reasons including auditing, provenance, transparency, accountability, introspective analysis, and backups [10, 17, 31, 35, 45, 52]. As a result, there has been an increasing interest in using *no-overwrite* or *immutable* data stores, where all data ever generated or produced is somehow persisted, either directly or in the form of *lineages* that can be used to reconstruct them.

Delta encoding is a cornerstone of many *no-overwrite* storage systems that are focused on archiving and maintaining vast quantities of datasets (simply put, a collection of files). Archival and backup systems often store multiple versions or snapshots of large datasets that have significant overlap across their contents using deltas. In version control systems, both for software (e.g., Git, SVN) and datasets (e.g., DataHub [10], noms [1]), it is common to store related file versions using this technique to save disk space while maintaining entire modification history. Delta encoding can result in remarkable improvements in storage requirements when compared to plain compression – in [11], the authors observe 65x reduction in storage space when just 100 snapshots of the Linux kernel repository were (individually) compressed using gzip vs using delta encoding.

At a high level, delta encoding consists of representing a *target* file content as the mutation, or delta, from a *source* file content. Intuitively, the source and target files are selected such that they have a large overlap across their contents and hence their delta is small. Furthermore, the source file itself may be represented as a delta from another file, and so on, creating a "graph" of files and deltas. The compressed storage is obtained by keeping only a few select files, commonly referred to as *materialized* files, and deltas (instead of the files they represent) in this graph, such that it is possible to re-create any file by walking the *path* of deltas starting from a materialized file and ending at the desired file.

Example 1. *Consider a toy* repository *containing two files that evolve as in Fig. 1(a). A node in this graph (called a* ***version graph****) represents a version or snapshot of the repository and an edge represents a derivation or transformation relationship between two versions. For instance, from V_1 to V_2, we see that file A_7 was modified to A_4 while A_1 remained unmodified. When available, information about such transformations are kept track of as edge meta-data.*

Fig. 1(b) shows how a delta-based storage system might store all files in all versions. Such a solution is typically meant to capture redundancies in the file contents across all versions of the repository

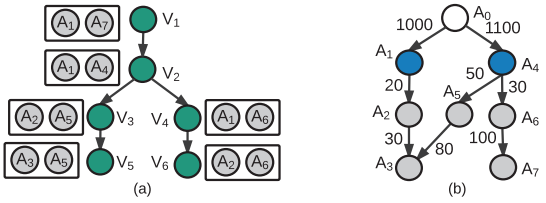


Figure 1: A delta encoding-based solution to store files, $A_1, \ldots, A_7$ across versions $V_1, \ldots, V_6$. The weights on the edges indicate delta sizes.

and ensure a reasonable trade-off between storage costs and access costs. We call this representation the ***storage graph****. Nodes in this graph represent files, and edges represent the deltas that are used to construct one file given another. For instance, an edge between A_2 and A_3 indicates that we can construct A_3 given A_2 and the respective delta. For algorithmic convenience, we include an empty file, A_0, as an entry point to traverse the storage graph. All files directly connected to A_0, i.e., A_1 and A_4, are* ***materialized files****, and are stored in their entirety. Further, the deltas, indicated by the edges, are also stored.*

Note that although a tree would be sufficient to guarantee that we can construct any file, the storage system may decide to include additional deltas to make co-accesses efficient; e.g., if A_3 is commonly accessed with A_2 and A_5 it would be beneficial to store it as a delta from both A_2 and A_5.

There has been significant work on various aspects of delta encoding-based storage systems: computing near-optimal deltas for a variety of data formats [20, 43], quickly finding ideal files to delta from [22], and supporting delta storage in file systems, scientific databases, network transport, etc. [39, 45]. However, existing delta-oriented storage engines offer limited or no support for querying the data stored within them; the primary query type supported by those engines is *checkout*, i.e., reconstructing a specific version of a dataset or a file. With such storage engines becoming efficient and mainstream, there is an increasing desire and opportunity to perform rich analysis queries over the historical information contained within such data stores [16]. The queries of interest include auditing or provenance queries over the datasets (e.g., identify the datasets where a particular property holds), analyzing the evolution of a dataset over time (i.e., temporal analytics), and comparing results of SQL-like queries over different versions of the same dataset (obtained through, e.g., applying different analysis pipelines to the same initial dataset). In general, combining *in situ* query processing [6] with the ability to query different versions of the same dataset can dramatically enhance the utility of immutable data stores.

However, delta-oriented storage engines of today require users to "check out" complete file/dataset versions in order to manipulate them. This approach is less than ideal particularly when the individual versions are large and the users need to access multiple versions for their analysis task. First, irrespective of the size of the query result, this approach entails creating all the input versions before query processing can begin, resulting in large memory and/or I/O usage. Second, it requires users to maintain another system to assist in executing the queries. Third, this approach fails to exploit the fact that most datasets evolve through changes that are small relative to the dataset sizes. Because the storage engine is aware of these properties, we argue that we can leverage this information to design computationally cheap methods to evaluate a query by *pushing down* query execution to the level of deltas. At first glance, this technique might be seen as an analogue to the notion of incremental view maintenance in relational databases [28]. However, a "query" in a relational database is typically defined on a single "version", whereas here we consider queries that span and reason about multiple versions simultaneously. In fact, the work on temporal analytics is closer in spirit than the work on incremental view maintenance.

In this paper, we initiate a systematic study of the problem of supporting rich analysis queries over delta-oriented storage engines, and describe the initial prototype of DEX, a *novel storage manager and query processing engine for delta-based storage*. Specifically, in this paper, we focus on the storage design and implementation of DEX for a class of semi-structured datasets that we call `datafiles`, and for a class of basic queries that includes multi-version checkouts, intersections, unions, and t-threshold queries. A `datafile` is a file whose contents can be seen as a *set* of records, i.e., the order of records within a `datafile` is immaterial, and no two records in a `datafile` are identical. Examples of such files include CSV files, JSON documents, log files, to name a few, and these constitute a large fraction of files in a typical data lake. A common method to represent a delta between two `datafiles` is to maintain the "deletions" and "additions" required to go from one `datafile` to the other. There is an inherent tension between the amount of information available in the deltas and the storage space they require, which directly impacts the types of queries that can be executed purely using the deltas. We chose the above delta format both because it is commonly used in practice, and because it strikes a good balance between storage space and query efficiency.

The current DEX prototype is built on top of `git`, a widely used source control system, analogous to how extensions like Git Large File Storage [2] are implemented. We provide an API, similar to `git`, to handle the standard version management tasks like *commit, checkout, status*, etc. The notable difference is that when a user starts tracking changes to a file (i.e., "adds" the file to the repository), she has an option to register the file as a `datafile`. Such files are stored by DEX using the techniques described in this paper, while all other files are managed by `git`.

Our key technical contributions are summarized as follows:

- To our knowledge, ours is the first work to systematically study how to optimize execution of different types of queries against delta-oriented storage engines.
- We develop a general cost-based optimization framework based on key *algebraic* properties regarding composition of the deltas. The result of this framework is a compact algebraic expression that confines query execution to a small set of deltas. As a side effect, the computational cost is dependent on the size and the number of deltas in the expression (which are typically small) in contrast to the size of the input datasets.
- We develop optimal algorithms for executing single-file or multi-file checkout queries assuming reasonable restrictions on the evaluation plan search space.
- We develop a series of intuitive transformation rules that help simplify the search space for intersection, union, and t-threshold queries, and use them in conjunction with cost-based solutions for base cases, to develop effective search algorithms.
- We have developed a prototype implementation of DEX on top of `git`, and we present a comprehensive evaluation against synthetic datasets of varying characteristics. Our results show that our methods perform exceedingly well compared to the baselines, even for simple queries like single-file checkouts.

2. SYSTEM OVERVIEW

We begin with a brief description of the user-facing data model and system architecture of DEX before describing the different types of queries that we support. Thereafter, we describe the system data model, i.e., the physical organization of data, and the primitives used by the system to evaluate the queries. While DEX can be integrated with any system that needs to store a large number of dataset

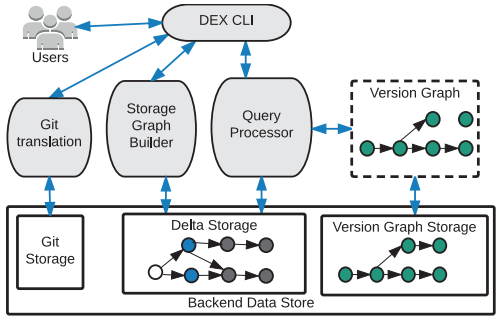


Figure 2: System Architecture of DEX; our focus in this work is largely on the design of the "Query Processor."

snapshots, in this work, we describe our models and associated terms in the context of a dataset version control system.

2.1 User Data Model

The user data model in DEX has two main abstractions – `datafile`, and *version* – that form the basis of all user interactions.

As mentioned above, a **`datafile`** is a file whose contents are interpreted as *set of records*. The user specifies a record separator when a `datafile` is added in the system. Within a `datafile`, we consider a record as an unstructured sequence of bytes. The only constraint we impose, however, is that a `datafile` cannot contain identical records: two records are said to be *identical* if they both have the same sequence of bytes. For instance, textual flat files such as CSV or logs can be seen as containing one record per line.

A **version** is a point-in-time snapshot of one or more `datafiles` typically residing in a directory on the user's file system. A version, identified by a unique ID, is immutable, and can be created at any point in time by any user who has access to the repository.

In addition to `datafiles` and versions, DEX also captures the version-level provenance – derivation and transformation relationships among the set of all versions – in a data structure called the **version graph**. Nodes in a version graph correspond to versions and edges capture relationships such as derivation, branching, transformation, etc, between two versions. One important use of this metadata is to allow rich queries over versions and provenance by means of any supported language/API (e.g., [16]). In this work, we do not limit ourselves to any particular API, but instead assume that we have an efficient method for finding all the `datafiles` referenced in a query. Since a version graph is typically much smaller than the `datafile` contents, it can be kept and traversed in memory to identify the versions that are referenced in a query.

We use the following notation to formalize the above discussion. Let $\mathcal{V}$ be the set of all versions. Each version $V \in \mathcal{V}$ contains a finite number of `datafiles`, say, $V = \{A_1, \ldots, A_t\}$. Let $\mathcal{A} = \{A_1, \ldots, A_n\}$ be the set of all `datafiles` across all versions. Note that it is possible for a `datafile` to be present in more than one version – this happens when the said `datafile` is not modified in the respective versions. The set of `datafiles` that appear in a version are kept track of as metadata in the corresponding node of the version graph. Let $A_a = \{r_1, \ldots, r_m\}$ be the set of records contained in `datafile` A_a. As mentioned before, no two records in a `datafile` are identical, i.e., $r_i \neq r_j, \forall r_i, r_j \in A_a$.

2.2 Queries

We now describe the semantics of each of the core operations that are the primary focus of this paper.

Checkout: *Checkouts* are the primary mechanism for reading off older versions of a dataset and it is imperative that a storage manager support them. Any version or any set of `datafiles` can be checked out, and the result is copied to the location suggested by the user (typically, it will be a directory on the user's machine). When a checkout query is issued, the version graph is consulted to identify the set of `datafiles` that comprise it. Specifically, the checkout operation takes as input a set of $k \geq 1$ `datafiles` $\mathcal{A}_k = \{A_{x_1}, \ldots, A_{x_k}\} \subset \mathcal{A}$ and outputs k files, one for each `datafile`. Henceforth, we use the notation $\textsc{Checkout}(\mathcal{A}_k)$ to denote the checkout operation.

Intersect: The *intersect* operation is an important operation when comparing the contents of a `datafile` that was modified across multiple versions. Similar to set intersection, given a set of $k \geq 2$ `datafiles` $\mathcal{A}_k = \{A_{x_1}, \ldots, A_{x_k}\} \subset \mathcal{A}$, the intersect operation outputs a single `datafile` containing records that appear in *all* `datafiles` in $\mathcal{A}_k$, i.e., $\{r : r \in A_{x_1} \wedge \cdots \wedge r \in A_{x_k}\}$. We use the notation $I(\mathcal{A}_k)$ to denote the intersect operation.

Union: The *union* operation, denoted by $U(\mathcal{A}_k)$, returns a single `datafile` containing records that appear in *any* of the `datafiles` in $\mathcal{A}_k$, i.e., $\{r : r \in A_{x_1} \vee \cdots \vee r \in A_{x_k}\}$.

***t*-Threshold:** Given as input a set of $k \geq 3$ `datafiles` $\mathcal{A}_k$ and an integer $1 < t < k$, the *t-threshold* operation, denoted by $T_t(\mathcal{A}_k)$, returns a single `datafile` that contains records appearing in *at least* t of the `datafiles` in $\mathcal{A}_k$. This generalizes the above operations – $t = 1$ and $t = k$ correspond to union and intersection respectively.

Although the above set of operations is intended as a starting point for investigating the nascent topic of query processing over deltas, these operations already enable many interesting queries. For example, comparing the results of intersection, union and/or t-Threshold across the versions of an evolving dataset can provide insights into the evolution process (e.g., properties of the records that change frequently vs those that remain static). Intersection or t-Threshold across the results of different machine learning pipelines on the same input dataset can help us identify which types of records are difficult to predict correctly, which can help an analyst steer the training process. Further, t-Threshold can return, for each record, a bitmap indicating the versions to which it belongs; depending on the semantics of the versions being queried, that information could be used for a variety of purposes including correlation analysis, anomaly detection, and visualizations. Finally, if specific analyses of interest are known in advance, materialized views (e.g., projections, results of aggregate queries or joins) can be computed in advance as the dataset versions are ingested; by exploiting the overlaps, these materialized views could be persisted cheaply in the storage engine itself. Although this requires a priori planning, the benefits at the time of querying could be tremendous. We plan to build support for defining and automatically materializing such views in future work, in addition to enriching the class of operations themselves.

2.3 System Architecture

The DEX prototype is built on top of `git` and has three major components: (a) a set of command line utilities, **DEX CLI**, written in Python, to allow the user to interact with the repository in the form of the standard *add, commit, checkout,* etc., commands (similar to `git`), (b) the **Storage Graph Builder** which decides how best to store a collection of `datafiles` (i.e., which deltas to use), and (c) the **Query Processor**, written in Java, that executes user queries against the deltas. `DEX CLI` passes through the version management tasks *not* pertaining to `datafiles` to `git`; the user may specify a file to be a `datafile` through a flag to the *add* command, and any tasks pertaining to those files are sent to the Storage Graph Builder (in case of *add* or *commit*) or the Query Processor.

The Storage Graph Builder performs tasks that primarily answer the question: When we have a collection of thousands of versions of `datafiles`, how to identify a good storage solution, i.e., decide

which datafiles to materialize and which datafiles to encode as deltas off of other datafiles? We encode this solution as an undirected weighted graph called the **storage graph** (§ 2.4.1).

In this work, our focus is on the Query Processor module, which accepts queries from users, uses the version graph to identify the datafiles referenced in the query, fetches appropriate deltas by analyzing the storage graph, and executes the queries.

The two data structures (storage and version graphs) as well as the deltas are persisted in a file system (other data stores like distributed key-value stores could also be used instead). Although the deltas themselves could be stored in a distributed fashion, in the current prototype, the Query Processor is designed to run in a single process. We expect this will be sufficient in most cases, since most queries are expected to touch only a small portion of the data; however, our techniques are easily parallelizable to handle large deltas.

2.4 System Data Model

Next, we discuss the storage graph and the *delta encoding* scheme used in DEX to store the versions on disk. Thereafter, we describe few properties of the deltas and discuss methods of combining them that will be useful in subsequent sections.

2.4.1 Storage Graph

Let $\mathcal{G} = (V, E)$ be a storage graph (see Fig. 3 for an example). Note that this graph is different from *version graph*, described in §2.1. While the version graph captures derivation or transformation relationships between versions of datasets, the storage graph represents information at the granularity of datafiles (encompassing all versions) and is meant to indicate delta relationships between them. Moreover, the storage graph is used by internal query execution routines and, unlike version graph, *is not intended to be exposed to the end user*. The vertex set V of the storage graph captures all unique datafiles across all versions, and a special *empty* datafile, A_0. Thus, $V = A_0 \cup \mathcal{A}$. The purpose of A_0 is to simplify many of algorithms that use the storage graph, both during its creation and during query evaluation (see [11] for a detailed usage).

An edge $e(A_i, A_j) \in E$ represents the delta between datafiles A_i and A_j, and the edge set E represents the deltas that are chosen to store all datafiles. The weight of the edge w_e represents the storage cost (size in bytes) of the delta. For an edge $e(A_0, A_i)$, w_e represents the storage cost of A_i in its entirety (i.e., A_i is **materialized**).

We require that $\mathcal{G}$ be a connected graph so that it is possible to reconstruct any of the datafiles in $\mathcal{A}$. Specifically, a path from A_0 to A_i indicates the materialized datafile (one following A_0 on the path) and the sequence of deltas to apply in order to recreate A_i. Thus, to store all the datafiles in $\mathcal{A}$, it is sufficient to store only the materialized datafiles in $\mathcal{G}$ and all the deltas in E.

Prior systems have made use of the storage graph representation [3, 11, 48, 49], albeit with different monikers, to model a delta based solution to store data versions. The storage graph also generalizes the *sequence-of-deltas* model where the versions are ordered according to a certain criteria, e.g., timestamp, file size, etc., and every version except the first is stored as a delta against the previous one. The sequence-of-deltas model, although conceptually simple, has the downside that the retrieval time grows linearly with the number of versions stored. The storage graph representation addresses this limitation by allowing multiple versions to be derived from one version. For instance, if we require that every datafile derives 3 datafiles not derived by others, we can pack approximately 80K datafiles and have a maximum delta sequence of length 10.

2.4.2 Delta Representations and Tradeoffs

A key question for a delta encoding-based storage engine is selecting the *delta variant*, i.e., the particular format/algorithm for computing the delta between two files. This is because different delta formats are appropriate for different types of files: a UNIX-style line-by-line diff is a common delta format for plain text files, while an XOR is more suited to numerical array-oriented data. Exploiting the structure in the data, if known, can often lead to better deltas (e.g., for XML [46], or relations [40]). Column-based deltas may be more appopriate when a large number of records are changed slightly, e.g., due to a schema change. Furthermore, a particular delta format may be *directed* or *undirected*: if a delta Δ between source file A and target file B is directed, it may only be used to recreate B given A, and not vice versa. An undirected delta between two files, on the other hand, can accept either file as source and recreate the other.

The desire to execute queries directly on deltas (as we propose in this work) brings another dimension to this choice. There is an inherent tension in the amount of information stored in a delta, and our ability to push query execution on to them. In this work, we pick one of the most commonly used delta formats suitable for record-oriented files, that offers a good balance between the storage space required and the ability to execute queries. In addition, one can also consider keeping additional information or indexes, together with the deltas, to speed up certain queries (analogously to the work on *in situ query processing* [6]). For instance, Bloom filters [13] on deltas can be used to prevent unnecessary searches when selecting records that satisfy a predicate, aggregate summaries on deltas [42] can be used to speed up certain classes of aggregate queries, and as shown in Appendix D, bitmaps can be used to create a filtered index to speed up the queries described in this paper. Understanding these tradeoffs for different types of data and query classes is a rich area for future work that we plan to pursue.

2.4.3 Set-backed Deltas and Properties

The delta format that DEX uses, called *Set-backed Deltas*, is an undirected delta format, similar to the standard UNIX line-by-line diff. A set-backed delta Δ between a source datafile A_i and a target datafile A_j, is a set of two datafiles, Δ^- and Δ^+, that correspond to "deletions" and "insertions" respectively. Δ^- is the set of records that are present in A_i but not in A_j, while Δ^+ is the set of records that are not present in A_i but present in A_j. Δ can also be used to reconstruct A_i from A_j by exchanging Δ^- and Δ^+.

In DEX, we require deltas to be *consistent* [24], i.e., a delta does not contain the same record in Δ^- and Δ^+. This does not preclude updates to a record, including schema changes, since an update can be recorded as deleting the old record and adding a new record.

Definition 2 (Consistent Delta). *A delta is said to be consistent if* $\Delta^- \cap \Delta^+ = \emptyset$.

Because datafiles and deltas are sets, we will often make use of the following three standard operations on sets – union ($\cup$), intersection ($\cap$) and difference ($-$). Continuing the example, when we use Δ to construct A_j from A_i we call this operation **patching** A_i using Δ, and denote it as $A_j = A_i \oplus \Delta$.

Definition 3 (Patch). $A_i \oplus \Delta = (A_i - \Delta^-) \cup \Delta^+$

Observation 4. *If* Δ *is consistent,* $A_i \oplus \Delta = (A_i - \Delta^-) \cup \Delta^+ = (A_i \cup \Delta^+) - \Delta^-$.

Next, we describe another important property of set-deltas, called *contraction*. Intuitively, delta contraction corresponds to combining two deltas into a single delta such that the new delta has the same effect as applying the individual deltas. Formally, if A_1, A_2, A_3 are three datafiles and $\Delta_1 = \Delta(A_1, A_2), \Delta_2 = \Delta(A_2, A_3)$, we use the patch operator as before to represent contraction as follows,

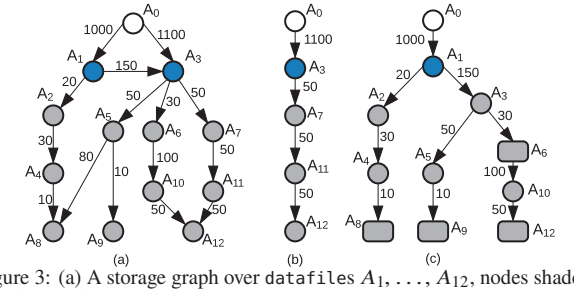


Figure 3: (a) A storage graph over `datafiles` $A_1, \ldots, A_{12}$, nodes shaded in blue (A_1, A_3) indicate materialized `datafiles`, edge annotations indicate the disk size of the delta; (b) access tree for $Q(A_{12})$, this is the shortest path from A_0 to A_{12}; (c) access tree for $Q(A_6, A_8, A_9, A_{12})$, this is the minimum cost Steiner tree for the terminals $\{A_0, A_6, A_8, A_9, A_{12}\}$

Definition 5 (Delta Contraction). $\Delta = \Delta_1 \oplus \Delta_2$, *where,*

$$\Delta^- = (\Delta_1^- - \Delta_2^+) \cup \Delta_2^-; \qquad \Delta^+ = (\Delta_1^+ - \Delta_2^-) \cup \Delta_2^+ \qquad (1)$$

Although delta contraction, as defined above, can be applied to two arbitrary deltas, the result is well-defined only if the target `datafile` of Δ_1 is same as the source `datafile` of Δ_2. The result Δ has the same source `datafile` as Δ_1 and derives the target `datafile` of Δ_2.

This definition can be generalized to a sequence of deltas: the contraction of a sequence of deltas $\Delta_1, \ldots, \Delta_m$ is the result of the operation $\Delta_1 \oplus \cdots \oplus \Delta_m$.

Given the above properties, we can infer that:

Observation 6. *If Δ_1 and Δ_2 are consistent, then their contraction, $\Delta = \Delta_1 \oplus \Delta_2$, is also consistent.*

Observation 7. *The patch operation is associative, i.e.,* $(\Delta_1 \oplus \Delta_2) \oplus \Delta_3 = \Delta_1 \oplus (\Delta_2 \oplus \Delta_3)$.

Although some of these observations might seem straightforward, formalizing them is crucial to argue the correctness of the transformations that we do later.

3. QUERY EXECUTION PRELIMINARIES

We begin with a more formal treatment of the query optimization problem, with first discussing the optimization metrics of interest and introducing the two-phase optimization approach that we take. We then briefly discuss the issues of cost and cardinality estimation and the search space of query evaluation plans.

Given a query, $Q(\mathcal{A}_k)$ where Q is one of $\{\textsc{Checkout}, I, U, T_t\}$ (§ 2.2) against a given storage graph $\mathcal{G}$, there are two somewhat independent stages in the overall query execution. First, we need to identify all the relevant `datafiles` and deltas in $\mathcal{G}$ that are necessary to execute $Q(\mathcal{A}_k)$. We refer to this problem as finding an *access tree* of $Q(\mathcal{A}_k)$, and describe it in detail in §3.2.

Second, given an access tree, we need to devise an efficient *evaluation plan*, that describes exactly what operations are used to compute the result of $Q(\mathcal{A}_k)$. This plan is represented as a *delta expression*: an algebraic expression where the operands are `datafiles` and deltas from the storage graph $\mathcal{G}$, and the operations are patch and primitive set operations. During this stage, we also consider the problem of finding a good ordering of evaluating the different operations in the delta expression. We describe the techniques for each query $Q \in \{\textsc{Checkout}, I, U, T_t\}$ in § 4.

3.1 Optimization Metrics

To be able to develop a systematic cost-based approach to query execution, we first need to identify appropriate *optimization metrics* and *cost models*. It is unfortunately difficult to develop a single cost metric that captures the costs of the two stages discussed above, which also makes it hard to do joint optimization across them. Because the backend store is likely to be relatively expensive to access (we expect it to be distributed in general), we would like to minimize the amount of data that is read from the backend store; this also reduces the network I/O. Once the data has been gathered, however, the different ways to evaluate a query can have very different CPU costs and wall-clock time. Hence, for the second phase, we would prefer to use a metric that tracks the CPU cost.

We adopt a two-phase approach in DEX inspired by this. We first find the best "access tree" that minimizes the total amount of data that needs to be read (in bytes) from the backend store. In other words, we identify the set of `datafiles` and deltas that have the smallest total size, that are sufficient to reconstruct the required `datafiles`. We then search for the best evaluation plan according to a cost model that estimates the CPU resources needed by the plan. We discuss the specifics in further detail in § 3.4 when we discuss the operator implementations.

We do not explicitly account for disk access costs during the second phase for several reasons. First, although the overall storage graph and the delta sizes in total are expected to be very large, the access tree for any given query is typically much smaller and the deltas constituting that will typically fit in the memory of a powerful machine. More importantly, most of our algorithms (§ 3.4) access the deltas sequentially (while reading and writing), and thus even if the deltas were disk resident (or intermediate results needed to be written to disk), the CPU and/or the memory bandwidth is still the main bottleneck. One exception here is binary search or gallop search (that an intersection operation might employ) where our approach might underestimate the cost of an intersection in case of extreme skew. However, our cost estimation procedure can be easily modified to account for that case. Moreover, the deltas are typically stored in a compressed fashion on disk, thereby making it necessary to uncompress them by reading them once into memory, and further making the overall computation CPU-bound.

3.2 Access Tree

Given a query $Q(\mathcal{A}_k)$, an access tree, $\mathcal{G}_Q = (V_Q, E_Q)$ is a subgraph of $\mathcal{G}$ such that: (i) $A_0 \cup \mathcal{A}_k \subseteq V_Q \subseteq V$, and (ii) $\mathcal{G}_Q$ is a tree, i.e., a connected graph with no cycles.

The first condition implies that all `datafiles` required by the query are part of the access tree. The second condition ensures that we have a *valid* and *minimal* solution: (i) Valid: because $\mathcal{G}_Q$ is connected, there exists at least one path between A_0 and A_{x_i}, which denotes the materialized `datafile` and the sequence of deltas to apply to reconstruct A_{x_i}, (ii) Minimal: because $\mathcal{G}_Q$ is a tree, for every $A_{x_i} \in \mathcal{A}_k$, $\mathcal{G}_Q$ contains exactly one path from A_0 to A_{x_i}.

We define the *cost* of an access tree as the sum of weights of all edges in it, i.e., $C(\mathcal{G}_Q) = \sum_{e \in E_Q} w_e$. When the edge weights correspond to the sizes of the deltas, this definition captures the cost metric mentioned above. To address the problem of identifying the least cost access tree, we consider two cases, $k = 1$ and $k > 1$. We refer to these as *single* `datafile` *access* and *multiple* `datafile` *access* respectively.

Single `datafile` Access: When $k = 1$, $\mathcal{A}_1 = \{A_{x_1}\}$. Any A_0 to A_{x_1} path in $\mathcal{G}$ satisfies the conditions of an access tree. Thus, finding the least cost access tree amounts to finding the shortest path between A_0 and A_{x_1}, and we use the classical Dijkstra's algorithm.

Multiple `datafile` Access: When $k > 1$, the problem of finding a low cost access tree is equivalent to finding a *Steiner Tree* [33]. Here, the set of nodes $A_0 \cup \mathcal{A}_k$ act as terminals and our objective is to find a minimum cost Steiner tree that contains all of them. This problem is $\mathcal{APX}$-Hard, i.e., arbitrarily good approximations cannot be achieved in polynomial time (unless $\mathcal{P} = \mathcal{NP}$). In this work, we

use the classical 2-approximation algorithm, which finds a tree with cost *at most* 2 times the optimal.

Example 8. *Consider the query* Checkout$(\{A_6, A_8, A_9, A_{12}\})$ *on the storage graph in Fig. 3(a). Fig. 3(c) shows the least cost access tree for this query.*

3.3 Search Space

Cost-based optimization requires us define the search space of potential, equivalent plans. The search space that we use in this work revolves around two equivalences: (i) associativity of the *patch* operation, and (ii) De Morgan's laws for set theory. We can thus generate equivalent evaluation plans by repeatedly applying those equivalence rules. Unfortunately the number of different evaluation plans is very large, even with just the first rule (§ 4.1). Unlike relational query optimization, the set of potential intermediate results is not easy to define either, and thus this problem does not seem amenable to dynamic programming-style algorithms used there. We instead take a hybrid approach where we use a series of heuristic transformation rules, based on De Morgan's laws, to simplify the expressions, and use a dynamic programming-based algorithm (that exploits the associativity of patch) to optimize the sub-expressions in the simplified expression.

Apart from generating alternative query expressions using logical equivalence rules, it is also possible to expand the search space of candidate plans by considering the impact of physical access structures on the data, e.g., secondary indexes. For instance, B-Trees on datafiles or deltas can be helpful when records are filtered on some attribute, bloom filters on deltas can help in evaluating queries like set difference, and so on. Additional considerations also arise when a join result is required across multiple versions – the delta chains for the different sets of datafiles (corresponding to the different relations) may not be "aligned" and the access tree selection will have to consider possibility of "joint" optimizations. Understanding this search space further, especially for richer queries involving joins and aggregates, remains a rich area for future work.

3.4 Cost and Cardinality Estimation

The cost of executing any of the set operations mentioned so far depends on the physical `datafile` format and the specific implementation of the operation. Since there exist several implementations for the set operations, there exist several cost functions. In DEX, the primary method of storing a `datafile` is *clustered storage*. In this method, records are stored in a sorted manner based on a suitable derived key (e.g., SHA1). There are several algorithms for evaluating a set expression between two or more operands based on this storage format and we outline our choices next alongwith their respective cost. To keep the discussion simple, we describe algorithms and their respective cost functions when all input data for a specific operation fits in memory and there is no paging of intermediate results to disk. Even if some deltas are large enough to require using disk, most of the algorithms below access the deltas sequentially and thus can be used with small modifications. We note that our optimization algorithms are largely agnostic to the specific choices for operator implementations, and can be used as long as the costs of the operations can be estimated.

Intersection: To compute the intersection of l `datafiles`, $A_1, \ldots, A_l$, we use an *adaptive* algorithm introduced in [19] called Small Adaptive (SA). SA first sorts the set of input `datafiles` according to their size. For each element in the smallest `datafile`, SA performs a *gallop search* on the second smallest `datafile`. A gallop search consists of two stages. In the first stage, we determine a range in which the element would reside if it were in the `datafile`. This range is found by identifying the first exponent j such that the element at 2^j is greater than the searched element. In the second stage, a binary search is performed in the range $(2^{j-1}, 2^j)$ to find if the element exists. If found, a new gallop search is performed in the remaining $l - 2$ `datafiles` to determine if the element is present in the intersection, otherwise a new search is performed. After this step, each `datafile` has an examined range (from the beginning to the position returned by the current gallop search) and an unexamined range. SA then selects two `datafiles` with the smallest unexamined range and repeats the process until one of the `datafiles` has been fully examined.

Because intersections only make sets smaller, as the algorithm progresses with several sets, the time to do each intersection effectively reduces. In particular, as pointed to in [19], the algorithm benefits largely if the set sizes vary widely, and performs poorly if the set sizes are all roughly the same. Since one gallop search takes $O(\log i)$ time, where i is the index where the element would be in the `datafile`, we can model the worst case cost of intersection as,

$$C_{\cap}(A_1, \ldots, A_l) = l|A_1| \log(|A_l|/|A_1|), \tag{2}$$

where A_1 and A_l are the smallest and largest `datafiles` respectively.

Union: To take a union of l `datafiles` $\{A_1, \ldots, A_l\}$, we perform a linear scan over all lists to merge them, and output the result.

$$C_{\cup}(A_1, \ldots, A_l) = |A_1| + \cdots + |A_l|. \tag{3}$$

Set Difference: To compute the set difference $A_1 - A_2$, we choose the better among the following two based on input sizes: perform a linear scan over both `datafiles` and use a merging algorithm, or for each element in A_1, perform a gallop search on A_2, including the element in the output if the search fails. This can be captured using the cost function,

$$C_{-}(A_1, A_2) = \min\{|A_1| + |A_2|, |A_1| \log(|A_2|/|A_1|)\}. \tag{4}$$

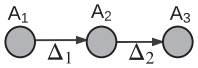
Patch: This is a binary operation where the two inputs are either (i) a `datafile` (M) and a delta (Δ), or (ii) two deltas (Δ_1 and Δ_2). In the first case, the output `datafile` can be computed by performing one linear scan over each of M, Δ^+ and Δ^- and evaluating Definition 3, making the cost function,

$$C_{\oplus}(M, \Delta) = |M| + |\Delta|. \tag{5}$$

Typically, $|M| > |\Delta^-|$, and we use the linear scan approach to compute the set difference. In the second case, the output Δ can be computed by evaluating Definition 5. Note that the `datafiles` of Δ_2 are scanned twice, once to compute Δ^+ and once to compute Δ^-. Thus, the cost function is given as,

$$C_{\oplus}(\Delta_1, \Delta_2) = |\Delta_1| + 2|\Delta_2|. \tag{6}$$

Cardinality Estimation: Because we restrict the search space as discussed in § 3.3, we require intermediate result size estimates only when two deltas are patched.



Let $\Delta = \Delta_1 \oplus \Delta_2$, where Δ_1 and Δ_2 are deltas between three `datafiles` as above. Let $x = |\Delta^-|$ and $y = |\Delta^+|$. We want to estimate x and y. By definition, Δ is a consistent delta between A_1 and A_3. Therefore, $|A_3| = |A_1| - x + y$. Since $|A_1|$ and $|A_3|$ are known, we can estimate x from y, or vice versa.

From Definition 5 we can obtain intervals for both x and y as,

$$x \in \left[\max(0, |\Delta_1^-| - |\Delta_2^+|) + |\Delta_2^-|, |\Delta_1^-| + |\Delta_2^-|\right],$$
$$y \in \left[\max(0, |\Delta_1^+| - |\Delta_2^-|) + |\Delta_2^+|, |\Delta_1^+| + |\Delta_2^+|\right].$$

We estimate the quantity with the smaller interval, where the value is chosen uniformly at random from the corresponding interval.

4. QUERY EXECUTION ALGORITHMS

Next we present a series of algorithms for cost-based optimization for each of the different query types.

4.1 Checkout Queries

Let $\textsc{Checkout}(\mathcal{A}_k)$ and $\mathcal{G}_Q$ denote a checkout query and its access tree resp. We first consider the case when $k = 1$ (single `datafile` checkout) followed by the case $k > 1$ (multiple `datafile` checkout).

4.1.1 *Single* `datafile` *Checkout*

Recall that the access tree $\mathcal{G}_Q$, when $k = 1$, is the shortest path from A_0 to A_{x_1} in $\mathcal{G}$. The delta expression for single `datafile` checkout is therefore, of the form, $Q : M \oplus \Delta_1 \oplus \Delta_2 \oplus \cdots \oplus \Delta_m$, where M is the materialized `datafile`.

Evaluation Algorithms: Since the $\oplus$ operation is associative, we can evaluate Q in multiple ways by changing the placement of "parentheses". For instance, one method is to evaluate the expression from left-to-right, i.e., $Q : (((M \oplus \Delta_1) \oplus \Delta_2) \oplus \cdots \oplus \Delta_m)$. Alternately, we can evaluate the expression from right-to-left, or in any arbitrary fashion that repeatedly combines two operands at a time, until we are left with the result. These evaluation methods, in general, will have varying costs. The total number of evaluation orders is equivalent to the classical problem of counting the number of ways of associating m applications of a binary operator, and is given by the $(m-1)$th Catalan number, which is $\Omega(4^m/m^{3/2})$.

Note that a greedy algorithm that iteratively combines two deltas having the least cost is not always the optimal strategy.

Example 9. *Consider the expression, $Q : \Delta_1 \oplus \Delta_2 \oplus \Delta_3$, where the deltas are such that $|\Delta_1| = x, |\Delta_2| \simeq |\Delta_3| = y, x \ll y$. Intuitively, the deltas Δ_2, Δ_3 are larger compared to Δ_1 and they are such that they almost "undo" each other. The greedy algorithm will pick the plan $(\Delta_1 \oplus \Delta_2) \oplus \Delta_3$ with estimated cost $\approx 2x + 5y$, while the optimal plan $\Delta_1 \oplus (\Delta_2 \oplus \Delta_3)$ has cost $\approx x + 2y + 2\varepsilon$, where $\varepsilon = |\Delta_2 \oplus \Delta_3|$.*

For sake of completeness, we have reproduced the classical dynamic programming algorithm to select the (estimated) best evaluation order in Appendix A.1. We call this the *path contraction (`PC`)* algorithm. We use `PC` extensively in subsequent sections to determine the best evaluation order to combine a sequence of deltas. The runtime of `PC` is $\Theta(m^3)$ where m is the number of deltas.

As discussed in § 3.3, we have syntactically restricted the space of alternative evaluation plans for checkout by only considering the associativity of the patch operation. Although additional transformations could be used to expand the search space, we could not identify any such transformation rules for checkout that were effective outside of pathological cases.

4.1.2 *Multiple* `datafile` *Checkout*

Since $\mathcal{G}_Q$ is a tree, there exists exactly one path from A_0 to each $A_i \in \mathcal{A}_k$. Let $\rho(A_i)$ denote the sequence of deltas on this path. A straightforward method to evaluate the query is to consider the delta expression for each A_i based on the deltas in $\rho(A_i)$ and use `PC` to get the optimal execution order. However, doing so does not take into account the opportunity for shared computation. Specifically, two or more paths may share sub-expressions and we end up evaluating a sub-expression multiple times if we consider each path independently. We illustrate this with the help of an example.

Example 10. *Consider the query $\textsc{Checkout}(A_6, A_8, A_9, A_{12})$ and the access tree in Fig. 3(c). We write one expression for each of A_6, A_8, A_9 and A_{12} respectively, as follows,*

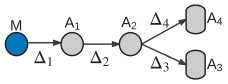
$$\begin{aligned} Q : & A_1 \oplus \Delta(A_1, A_2) \oplus \Delta(A_2, A_4) \oplus \Delta(A_4, A_8); \\ & A_1 \oplus \Delta(A_1, A_3) \oplus \Delta(A_3, A_5) \oplus \Delta(A_5, A_9); \\ & A_1 \oplus \Delta(A_1, A_3) \oplus \Delta(A_3, A_6); \\ & A_1 \oplus \Delta(A_1, A_3) \oplus \Delta(A_3, A_6) \oplus \Delta(A_6, A_{10}) \oplus \Delta(A_{10}, A_{12}) \end{aligned}$$

Note that if we evaluate each of these independently, based on how the parenthesization is performed, we will evaluate $A_1 \oplus \Delta(A_1, A_3)$ thrice, or $\Delta(A_1, A_3) \oplus \Delta(A_3, A_6)$ twice.

Evaluation Algorithms: The above can be seen as the problem of how to plan the execution of a batch of queries, where each query is a single `datafile` checkout, analogous to multi-query optimization. The goal here is to design a strategy that recognizes the possibilities of shared computation so that we can re-use the result of sub-expressions to the extent possible in order to obtain a globally optimal evaluation plan. To that effect, we develop a dynamic programming algorithm, called *tree contraction (TC)*, to select the best evaluation plan after accounting for shared computation. At a high level, `TC` breaks up the problem into two questions: how do we decide which sub-expressions to share and how do we best parenthesize each (sub-)expression? We already know how to compute the solution for the latter using `PC`. The solution to the former is based on enumerating all possibilites for sub-expression sharing and recursively solving the rest of the problem with the help of the extra state information computed during `PC`. The pseudocode of `TC` is shown in Appendix A.2, and its time complexity is $O(m^3)$ where m is the number of deltas in the access tree.

Before we conclude this discussion, it will be helpful to understand, as the following example shows, why a simple greedy strategy of always sharing the largest possible expression (from left to right) is not always optimal.

Example 11. *Consider the following access tree to checkout A_3 and A_4.*



The instance is constructed such that $\Delta_2, \Delta_3, \Delta_4$ are large (say, $y \approx |\Delta_2| \approx |\Delta_3| \approx |\Delta_4|$) and Δ_3, Δ_4 "undo" most of the changes done by Δ_2. Δ_1 is a small independent set of changes, say, $x = |\Delta_1|, x \ll y$. The greedy strategy will force us to share $\Delta' = \Delta_1 \oplus \Delta_2$ and $|\Delta'| \approx x + y$. Thus the cost of the greedy strategy is $\approx 3x + 8y$. On the other hand, evaluating $\Delta_1 \oplus (\Delta_2 \oplus \Delta_3), \Delta_1 \oplus (\Delta_2 \oplus \Delta_4)$ incurs a cost $\approx 2x + 6y$.

4.2 Intersection Queries

Given an intersect query $I(\mathcal{A}_k)$ and its access tree $\mathcal{G}_Q$, a straightforward method, that we treat as a baseline, is to first use `TC` to perform $\textsc{Checkout}(\mathcal{A}_k)$ followed by the intersection. This approach, however, only considers the associativity of patch and the sharing of sub-expressions in order to find a good evaluation order. We now develop a set of *transformation rules* on the access tree that allow us to compute *partial intersection results* using only the deltas. Since a delta between two `datafile`s already captures a notion of difference between them, we leverage this information and avoid redundant computation while finding the intersection.

The transformation rules are based on identifying two simple structures in the access tree $\mathcal{G}_Q$, called *line* and *star* (Fig. 4). In each figure, we use boxes to denote `datafile`s in $\mathcal{A}_k$ and circles to denote other `datafile`s. Also, if a box or circle is filled, it denotes a materialized `datafile`.

Line Access Trees: Consider the query $I(A_1, A_2)$ with the `datafile`s as arranged in Fig. 4(a). Here, A_1 is the materialized `datafile`

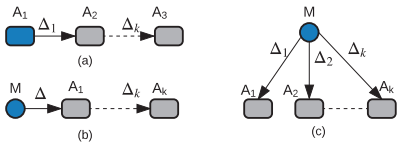


Figure 4: (a) A line of two or more `datafiles`; (b) A line when the materialized `datafile` M is not a part of query input; (c) A star.

while A_2 is stored as a delta from A_1. It is easy to see that:

$$R = A_1 \cap A_2 = A_1 - \Delta_1^-.$$

In general, for the query $I(\mathcal{A}_k)$ with the `datafiles` as arranged in Fig. 4(a), the result R is computed as,

$$R = I(\mathcal{A}_k) = A_1 - (\Delta_1^- \cup \cdots \cup \Delta_{k-1}^-) \tag{7}$$

Note that the above equality does not hold if there are other `datafiles` in $\mathcal{G}_Q$ even if $\mathcal{G}_Q$ is a line. We use this equality to introduce our first transformation rule that "reduces" the deltas in a line structure to a single delta that gives the result for the intersect query. Conceptually, this reduced delta acts as a delta between two `datafiles`: the same materialized `datafile` as in the line and a (new) `datafile` representing the intersection result.

T1:– If $\Delta_1, \ldots, \Delta_{k-1}$ are the deltas in the line, then the reduced delta, Δ_l, for the intersect query is composed as,

$$\Delta_l^- = \Delta_1^- \cup \Delta_2^- \cup \cdots \cup \Delta_{k-1}^-; \quad \Delta_l^+ = \emptyset$$

This transformation rule significantly reduces the amount of data that needs to be subsequently processed.

To handle the case when the materialized `datafile` is not a part of the line, as in Fig. 4(b), we use a two-step approach. First, assuming that A_1 is the materialized `datafile` and we can use rule **T1** to compute the reduced delta Δ_l. Second, we can contract Δ and Δ_l since they share the `datafile` A_1. The result is computed as $R = M \oplus \Delta \oplus \Delta_l$.

Next, we discuss how to evaluate eq. (7). Consider the identity, $X - (Y \cup Z) = (X - Y) - Z$, for three sets X, Y, Z. If $|Y|, |Z| \ll |X|$, observe that $X - (Y \cup Z)$ will often have less cost than $(X - Y) - Z$. Intuitively, if we do the set difference first, then $|X - Y|$ will be comparable to $|X|$ and will end up being scanned again. Specifically, under the cost model stated in § 3.4, when $|X| > 3\max(|Y|, |Z|)$, performing $X - (Y \cup Z)$ will result in a reduced cost. We therefore use the following greedy heuristic when evaluating eq. (7).

H1:– Let $\mathcal{L} = \{\Delta_1^-, \ldots, \Delta_{k-1}^-\}, R = M$. We iteratively perform the following until $\mathcal{L}$ is empty: let Δ' be the largest size delta in $\mathcal{L}$. If $|R| > 3|\Delta'|$, we replace the largest two deltas in $\mathcal{L}$ by their union; else, we set $R = R - \Delta'$.

Star Access Trees: Consider the query $I(A_1, A_2)$ with the `datafiles` as arranged in Fig. 4(c). Here, M is the materialized `datafile` and A_1 and A_2 are stored as deltas from M. We have that:

$$R = A_1 \cap A_2 = \left(M - (\Delta_1^- \cup \Delta_2^-)\right) \cup (\Delta_1^+ \cap \Delta_2^+)$$

To see why, recall that Δ_i^- indicates the set of records to be removed from M to get A_i. Hence, no record in Δ_i^- can be a part of the intersection result. Additionally, new records (that do not exist in M) can be added only if they belong to all of Δ_i^+.

In general, for the query $I(\mathcal{A}_k)$ with the `datafiles` as arranged in Fig. 4(c), the result R is computed as,

$$R = I(\mathcal{A}_k) = \left(M - (\cup_{i=1}^k \Delta_i^-)\right) \cup (\cap_{i=1}^k \Delta_i^+) \tag{8}$$

The result R is written in terms of the materialized `datafile` M. This leads us to our second tarnsformation rule that "reduces" the deltas in a star structure to a single delta that gives the result for the intersect query. Conceptually, this reduced delta acts as a delta between M and the intersection result.

T2:– If $\Delta_1, \ldots, \Delta_k$ are the deltas in the star, then the reduced delta Δ_s, for the intersect query is composed as,

$$\Delta_s^- = \cup_{i=1}^k \Delta_i^-; \quad \Delta_s^+ = \cap_{i=1}^k \Delta_i^+$$

We use **H1** to evaluate Equation (8). Since none of Δ_i^+ can help reduce intermediate result sizes, the intersection of Δ_i^+s can be done independently. Finally, we also make the following observation.

Observation 12. *Δ_l and Δ_s are consistent.*

Arbitrary Access Trees: We develop an algorithm, called *Contract and Reduce (C&R)*, that puts the above two techniques together for arbitrary access trees. With minor modifications, the same algorithm can be used for other types of queries, and hence we describe its general form. The pseudocode for C&R is shown in Appendix A.3.

Starting with a query $Q \in \{I, U, T_t\}$, and its access tree $\mathcal{G}_Q$ as inputs, C&R iteratively evaluates partial delta expressions, effectively reducing the size of $\mathcal{G}_Q$. Each iteration of the algorithm has two phases: *contract* phase and *reduce* phase. In the contract phase, we identify all *maximal continuous* delta paths: a path where all nodes, except the start and end node, have exactly 2 neighbors, and none of the intermediate nodes is a part of $\mathcal{A}_k$. Each path should be of length > 2 and be the longest possible. Every such path is then contracted to a single delta using PC. Specifically, if $\Delta_1, \ldots, \Delta_u$ is the sequence of deltas on the path between two nodes A_x and A_y in $\mathcal{G}_Q$, we use PC to find the best order to evaluate $\Delta_\rho = \Delta_1 \oplus \cdots \oplus \Delta_u$, execute the operations, and replace the sequence by the delta Δ_ρ between A_x and A_y.

In the reduce phase, we find all lines and stars in $\mathcal{G}_Q$ and reduce them according to the appropriate transformation rules – **T1/T3** for lines and **T2/T4/T5** for stars. Each transformation takes as input two or more deltas, either in a line or star configuration, and replaces them by a single delta. Note that if all paths are contracted, and number of deltas in $\mathcal{G}_Q$ is more than 1, there will at be at least one reduction to be performed.

The algorithm ends when there is only one delta remaining in $\mathcal{G}_Q$. At this point, we simply apply the delta to the materialized `datafile` in $\mathcal{G}_Q$ and return the result. We illustrate the behaviour of the algorithm with the help of an example in Appendix A.3.

4.3 Union

In this section, we give transformation rules for *line* and *star* for the query $U(\mathcal{A}_k)$. We can then use C&R with the mentioned rules to evaluate arbitrary access tree structures.

Line: Consider the query $U(\mathcal{A}_k)$ with the `datafiles` as arranged in Fig. 4(a). Then: $R = A_1 \cup (\cup_{i=1}^k \Delta_i^+)$.
The transformation rule for a line can therefore be stated as,

T3:– If $\Delta_1, \ldots, \Delta_{k-1}$ are the deltas in the line, then the reduced delta, Δ_l, for the union query is composed as,

$$\Delta_l^- = \emptyset; \quad \Delta_l^+ = \cup_{i=1}^k \Delta_i^+$$

Star: If the `datafiles` are arranged as shown in in Fig. 4(c), we have that: $R = U(\mathcal{A}_k) = \left(M - (\cap_{i=1}^k \Delta_i^-)\right) \cup (\cup_{i=1}^k \Delta_i^+)$.
To see why, since Δ_i^- indicates the set of records to be removed from M to get A_i, if a record is absent in the union, it must have been present in all Δ_i^-. New records that are added in any Δ_i^+ are a part of the union result. Then:

T4:– If $\Delta_1, \ldots, \Delta_k$ are the deltas in the star, then the reduced delta Δ_s, for the union query is composed as,

$$\Delta_s^- = \cap_{i=1}^k \Delta_i^-; \quad \Delta_s^+ = \cup_{i=1}^k \Delta_i^+$$

We conclude this section by mentioning that similar to the intersection case, Δ_l and Δ_s, for the union query, are consistent.

4.4 t-Threshold

In order to evaluate a t-threshold query $T_t(\mathcal{A}_k)$, we make use of multiset-backed deltas during intermediate query execution, instead of the set-backed deltas that we have been using so far. This introduces two important issues during the execution of C&R that are the main focus of this section. First, we need to re-define the semantics of delta contraction in this new setting. Second, a line cannot be reduced in a straightforward manner as before. We begin with some definitions, describe the transformation rule for a star, and then discuss each of two issues in detail.

A multiset, unlike a set, allows multiple instances of any of its elements. We represent a multiset as $A = \{(r, c) : c \in \mathbb{N}_{\geq 1}\}$, where r is an element and $\mathbb{N}_{\geq 1}$ is the set of natural numbers. The number c is referred to as the multiplicity of r. A set is a multiset with all multiplicities as 1.

Consider the following two operations concerning multisets.

Definition 13 (Multiset Union). *Multiset union, denoted by $A = A_1 \uplus A_2$, returns a multiset containing elements that occur either in A_1 or A_2, where A_1 and A_2 are either multisets or sets. The multiplicity of an element in A is the sum of its multiplicites in A_1 and A_2.*

Definition 14 (Multiset Restrict). *If A is a multiset, $A_{c \leq p}$ is the set of elements in A with multiplicity at most p. Similarly, $A_{c \geq p}$ is the set of elements with multiplicity at least p.*

We now describe how to evaluate $T_t(\mathcal{A}_k)$ when $\mathcal{G}_Q$ is a star.

Star: Consider the query $R = T_3(\mathcal{A}_4)$, i.e., find all records that appear in at least 3 of $\{A_1, A_2, A_3, A_4\}$, when they are arranged in a star (Fig. 4(c)). Suppose the delta Δ is such that $\Delta^- = \Delta_1^- \uplus \Delta_2^- \uplus \Delta_3^- \uplus \Delta_4^-$ and $\Delta^+ = \Delta_1^+ \uplus \Delta_2^+ \uplus \Delta_3^+ \uplus \Delta_4^+$. Here, Δ^- and Δ^+ are multisets, and Δ is a multiset-backed delta. Then:

$$R = T_3(\mathcal{A}_4) = (M - \Delta^-_{c \geq 2}) \cup \Delta^+_{c \geq 3}$$

To understand why, consider a record $(r, c) \in \Delta^-$. This indicates that $r \in M$ and c of 4 deltas, $\{\Delta_1^-, \Delta_2^-, \Delta_3^-, \Delta_4^-\}$, ask to delete r. So long as $c \geq 2$, r will absent from at least 2 of $\{A_1, A_2, A_3, A_4\}$. Similarly, consider a record $(s, c) \in \Delta^+$. This indicates that $s \notin M$ and c of 4 deltas, $\{\Delta_1^-, \Delta_2^-, \Delta_3^-, \Delta_4^-\}$, ask to add s. So long as $c \geq 3$, k will be present in at least 3 of $\{A_1, A_2, A_3, A_4\}$.

More generally, the transformation rule for a star can be stated as below.

T5:– If $\Delta_1, \ldots, \Delta_k$ are the deltas in the star, then the reduced delta Δ_s, for the t-threshold query is composed as,

$$\Delta_s^- = \uplus_{i=1}^k \Delta_i^-; \quad \Delta_s^+ = \uplus_{i=1}^k \Delta_i^+$$

We note the following: (i) With every multiset-backed delta, we keep an integer value $2 \leq p(\Delta) \leq k$ which indicates the number of datafiles in $\mathcal{A}_k$ that are reduced by this delta. For instance, $p(\Delta) = 4$, in the previous example. In general, the access tree will have several disjoint stars (or lines) which are reduced at different times in the evaluation process. We discuss how $p(\Delta)$ is used shortly. (ii) The deltas $\Delta^-_{c \geq k-t+1}$) and $\Delta^+_{c \geq t}$ are consistent.

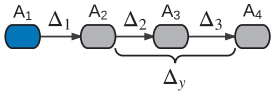
We now describe the semantics of the patch operation in the presence of multiset deltas.

Delta Contraction: Algorithm 4, shown in Appendix A, computes the result of $\Delta = \Delta_x \oplus \Delta_y$, where Δ_y is a multiset delta. Note that due to the nature of C&R, only the last delta in any sequence of deltas can be a multiset delta. The result delta Δ is also a multiset delta.

The main idea here is to preseve semantics of two values: (i) the multiplicity c of a record r, and (ii) the number of datafiles that are reduced due to the delta, $p(\Delta)$. Since this is a patch operation, we can simply set $p(\Delta) \leftarrow p(\Delta_y)$. Recall that a record $(r', c') \in \Delta_y^-$ indicates that c' of $p(\Delta_y)$ deltas ask us to remove r'. Now consider a record $r \in \Delta_x^+$. If $r \notin \Delta_y^-$, then we can add $(r, p(\Delta_y))$ to Δ^+. However, if $r \in \Delta_y^-$, then we need to "fix" the multiplicity of r, i.e., add $(r, p(\Delta_y) - c)$ to Δ^+, where c is the multiplicity of r in Δ_y^-. The other case is similar.

We use Algorithm 4 in place of the patch operator defined in §3.4 when one of the operands is a multiset delta. Its estimated cost is modeled as, $C_\oplus(\Delta_x, \Delta_y) = |\Delta_x| + 2|\Delta_y|$, and we can use PC during the contract phase as before.

Line: Consider the query, $R = T_2(\mathcal{A}_4)$, with the datafiles as shown below.



Consider a record r such that $r \in \Delta_1^+$, and $r \in \Delta_3^-$. Although r is present in two opposite deltas, $r \in R$. More generally, in the case of t-threshold queries, simply knowing whether a record is in Δ_i^- or Δ_i^+ is not sufficient to conclude if it is present in the result. We also require knowledge of the "position" of the delta containing the record on the line. Alternately, we can reduce the line by considering deltas in right-to-left order, by the following simple modification to Algorithm 4. Suppose that we know how to contract $\Delta_2 \oplus \Delta_3$ to obtain a multiset delta Δ_y as shown. We show how to modify Algorithm 4 to compute $\Delta = \Delta_1 \oplus \Delta_y$. The central idea is again to set record multiplicites and $p(\Delta)$ correctly. Note that $p(\Delta_y) = 2$, as it reduces A_3 and A_4. Since, Δ is also meant to reduce A_2, we set $p(\Delta) = p(\Delta_y) + 1$ (line 1). Consider a record $r \in \Delta_1^+$. If $r \notin \Delta_y^-$, then we can add $(r, p(\Delta_y) + 1)$ to Δ^+ (line 6). On the other hand, if $r \in \Delta_y^-$, we add $(r, p(\Delta_y) - c + 1)$ to Δ^+ (line 4) where c is the multiplicity of r in Δ_y^-. The other case is similar.

5. EXPERIMENTAL EVALUATION

In this section, we present a comprehensive evaluation of our DEX prototype. The key takeaway from our study is that, pushing down computation to the deltas can lead to signifincant savings, an order-of-magnitude in many cases. Surprisingly, even for a single datafile checkout, we see large benefits in the computational time. We also show, through an illustrative experiment (Appendix D), that using auxiliary data structures like bitmaps can increase the benefits many-fold, indicating that this is a rich direction for future work.

All experiments were conducted on a single machine with Intel Core i7-4790 CPU (3.60 GHz, 8MB L3 cache), 32GB of memory, running Ubuntu 16.04 and OpenJDK 64-bit server JVM (ver. 1.8.0_111). Our choice to write the query processor in Java was primarily based on getting quick development time while still being reasonably performant on large datasets. While using a low-level language (e.g., C) will reduce the absolute query execution times, it will not change our primary objective which is to measure relative speedup of our techniques compared to the baseline. All time measurements are recorded as wall-clock time. Unless otherwise stated, to measure response time, we run each query 10 times and consider the median. To account for the adaptive performance of some of the set operations, we repeat the above on 25 datasets with identical properties (described next) and report the median. As discussed in § 3.4, our computations are CPU bound, and we did not find an appreciable difference in warm cache vs cold cache settings; for consistency, we report results for a warm cache setting.

Datasets: Lacking access to real-world versioned datasets with sufficient and varied structure, we instead developed a synthetic data generator to generate datasets with very different characteristics for a wide variety of parameter values. This enables us to carefully study

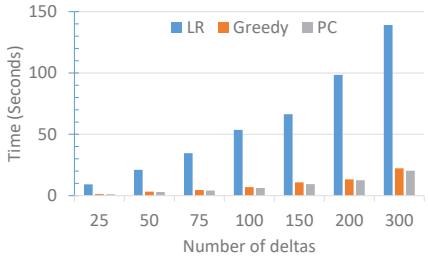


Figure 5: Effect of varying #Δ when |Δ| = 5%

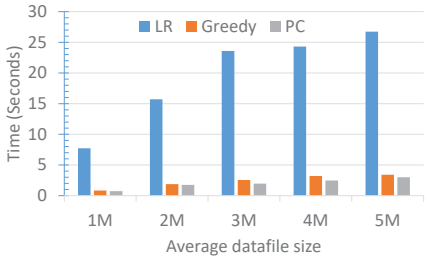


Figure 6: Effect of varying |A|

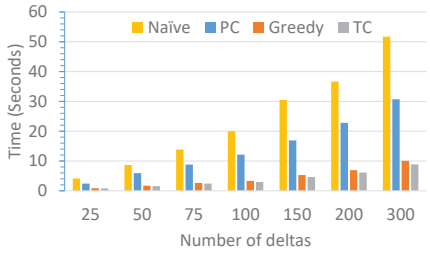


Figure 7: Effect of varying #Δ when |Δ| = 5%

the performance of our techniques in various settings. Formally, every experiment setting is characterized by a 4-tuple, $\langle T, |A|, |\Delta|, \#\Delta \rangle$, where $|A|$ and $|\Delta|$ refer to the average number records in a `datafile` and average size of the deltas in the dataset (as a percentage of $|A|$). T denotes the shape of the access tree that is used, and is one of: *line-shaped* (l), *star-shaped* (s) and *line-and-star* (ls); and #Δ refers to the number of deltas in the access tree. All records are 64-byte randomly generated strings. For more details on dataset generation process, please see Appendix B.

Single `datafile` Checkout: We begin with evaluating the performance of `PC`, i.e., Algorithm 1, against two heuristics for the case of single `datafile` checkout. Fig. 5 shows the median response time of this analysis (in milliseconds) on the vertical axis, and the horizontal axis is the number of deltas (#Δ) in the expression. The other parameters of the dataset are fixed at $\langle T = l, |A| = 3M, |\Delta| = 5\% \rangle$.

The `LR` heuristic simply evaluates the delta expression from left-to-right starting with the materialized `datafile`. This is the standard heuristic used in prior delta-based storage engines, like `git`. On the other hand, the Greedy heuristic iteratively patches two operands having the least estimated cost.

We observe that in each instance, `PC` performs better than Greedy which performs better than `LR`. Specifically, we note up to **7.0-8.8X** improvement in median response times when comparing `PC` with LR and up to 14% improvement when comparing with Greedy. The performance gap between `LR` and the other methods also increases slightly as the number of deltas goes up. This is because the left input of every patch operation in `LR` has a large size, in contrast to both Greedy and `PC`, that "balance" their inputs in a cost-based manner. Also, because we assume that every record in a `datafile` is equally likely to be modified and there is no set of "hot" records, i.e., records that are modified often, we observe that the intermediate result sizes continue to grow in Greedy and `PC` as well. We observe similar trends for other delta sizes and omit their results.

Next, we study the effect of varying average `datafile` size on the response times. Fig. 6 shows the result of this study on the dataset $\langle T = l, |\Delta| = 1\%, \#\Delta = 100 \rangle$ when $|A|$ is varied from 1 million records to 5 million records. In this case, we observe a **8.9-10.5X** speedup when compared to `LR`, with the Greedy solution being approximately close to `PC`.

Finally, although `PC` has cubic time complexity in the number of deltas, the solutions it finds are, in all cases, better than alternatives even after taking optimization time into consideration. When #Δ = 100, the average time to find the optimal solution was 1.2ms.

Multiple `datafile` Checkout: We now evaluate the time taken to checkout $k = 8$ `datafiles` on the dataset $\langle T = ls, |\Delta| = 5\%, |A| = 1M \rangle$. We evaluate `TC`, i.e., Algorithm 2, by comparison against three approaches. The Naive approach simply performs a checkout of each `datafile` independently using `LR`. The second approach uses `PC` to checkout individual `datafiles`. Both these approaches do not take into account sharing of intermediate results. The third approach, called Greedy, shares the results of the largest sub-expressions as much as it can (e.g., for two `datafiles`, the result of the expression from the root of the access tree to their lowest common ancestor is always shared).

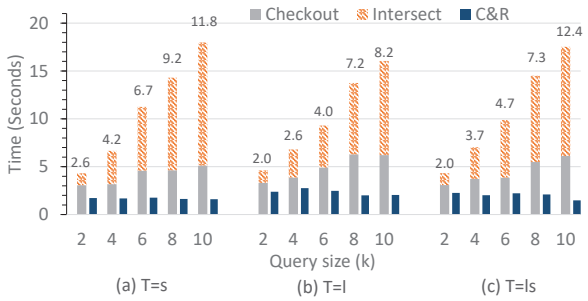


Figure 8: Effect of access tree structure when |Δ| = 1%, #Δ = 100

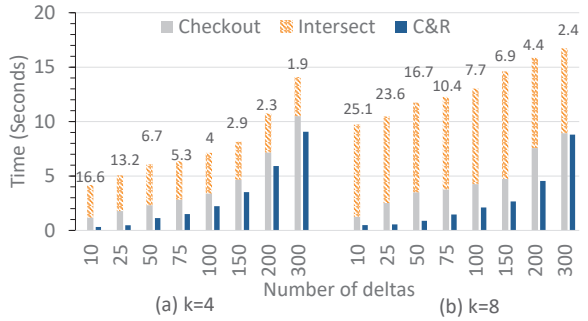


Figure 9: Effect of query size when |Δ| = 1%

Fig. 7 reports the median checkout time (in seconds) as the number of deltas (#Δ) in the access tree is varied. We observe that overall Greedy and `TC` have similar response times and `TC` performs slightly better than Greedy in each case (between 7.2 – 10.8% improvement). Also, when compared to Naive, we observe a **5.1–6.8X** improvement in median response time.

The average optimization time when #Δ = 300 was 18.4ms.

Intersect: In the following set of experiments, we compare the running time of evaluating $I(\mathcal{A}_k)$ using two algorithms. The baseline approach simply performs a checkout of all the `datafiles` in $\mathcal{A}_k$ using `TC`, followed by their intersection. The second approach measures the performance of `C&R`, i.e., Algorithm 3.

Because `C&R` makes decisions based on the shape of the access tree, we first study the effect of varying the shape of the access tree on intersect performance. Fig. 8 shows the median response time against the query size for the three types of access trees: line, star, and line-and-star. The other parameters of the dataset are $\langle |A| = 3M, |\Delta| = 1\%, \#\Delta = 100 \rangle$. The numbers on top of each bar indicate the speedup obtained. We note speedups of upto **12X** when using `C&R`. The speedup obtained for $T = l$ is smaller than others

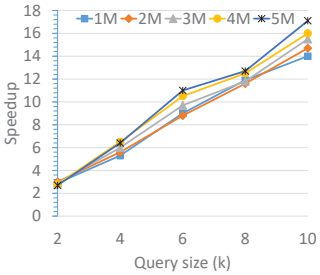


Figure 10: Intersect – Effect of $|A|$

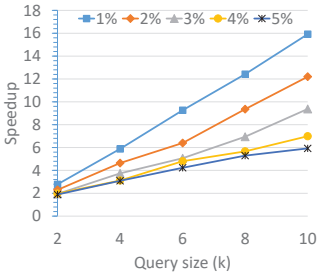


Figure 11: Intersect – Effect of $|\Delta|$

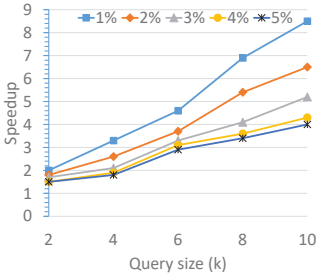


Figure 12: Union – Effect of $|\Delta|$

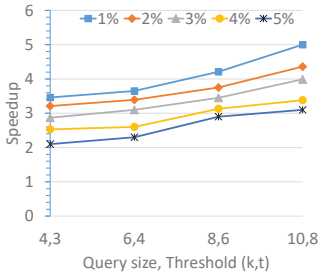


Figure 13: t-Thres. – Effect of $|\Delta|$

primariliy due to the shape of the access tree – in a line, the smallest path between root and a query `datafile` cannot be reduced using any of the tranformation rules and must be contracted using `PC`.

In the next experiment, reported in Fig. 9 we study the effect of varying the number of deltas in the access tree of $I(\mathcal{A}_k)$. Here, we use the dataset $\langle T = ls, |A| = 3M, |\Delta| = 1\% \rangle$ and vary #Δ; we report the results for $k = 4, 8$. As we can see, our techniques are particularly effective, giving a speedup upto **16X** and **25X**, for $k = 4$ and $k = 8$ respectively. The speedup decreases as the number of deltas increases primarily due to larger intermediate delta sizes.

Fig. 10 shows the speedup obtained when the average `datafile` size is varied between 1M and 5M records; other dataset parameters are $\langle T = ls, |\Delta| = 1\%, \#\Delta = 50 \rangle$. We observe that our techniques show significant benefit, obtaining upto **17X** speedup. Further, we note that `datafile` size does not affect `C&R` to a large degree.

Fig. 11 reports the speedup obtained when the average delta size in the dataset, $|\Delta|$, is varied between 1% and 5% of the average `datafile` size; other dataset parameters are $\langle T = ls, |A| = 3M, \#\Delta = 50 \rangle$. We note a speedup of **2.8-16X** when $|\Delta| = 1\%$ that decreases gradually to **2-6X** when $|\Delta| = 5\%$. This confirms our hypothesis that if the deltas between the `datafiles` are small, significant improvements can be obtained by using the deltas in query execution in a more direct manner. When the deltas get large, the intermediate result sizes grow too, which results in a reduced speedup.

Union: The results for $U(\mathcal{A}_k)$ are similar to the intersection case although with smaller speedup values. We report one such result in Fig. 12: the effect of varying query size (k) for datasets with different average delta size $|\Delta|$. The other parameters of the dataset are $\langle T = ls, \#\Delta = 50, |A| = 3M \rangle$. We note a speedup of **1.6-8.6X** when $|\Delta| = 1\%$ that decreases gradually to **1.5-4.1X** when $|\Delta| = 5\%$.

t-Threshold: We use the adaptive algorithm of [8] as a baseline for our t-threshold experiments. Similar to adaptive set intersection, this algorithm uses gallop search in order to find the position of an element r in a set. Moreover, it maintains a min heap of size $k - t + 1$, containing at most one element per set, in order to select a "good" element to probe other sets during each iteration. Fig. 13 reports the effects of varying (k, t) across datasets with different delta sizes. The other parameters of the dataset are $\langle T = ls, \#\Delta = 50, |A| = 3M \rangle$. We observe a speedup of **3.5-5X** when $|\Delta| = 1\%$ that gradually reduces as the delta size increases. When $|\Delta| = 5\%$, we report a speedup of **2.1-3.1X**. The overall speedup in this case is less than that obtained in the intersection or union query because unlike the two, the size of the intermediate results does not decrease when transforming lines and stars.

6. RELATED WORK

Queries in delta-based storage: Delta encoding has been used in a variety of systems to provide trade-offs among time, space, and compression performance, e.g., to reduce data transfer time for text/HTTP objects [41], to reduce access time in a file system [39], to store many versions of the generated artifacts in source code control systems (e.g., `git`) or other types of data [10, 40, 48]. Recently, Bhattacherjee et al. [11] provided a principled study of the trade-off between storage costs and access costs for many of the above schemes. However, the focus of many of the existing delta encoding schemes has been to access the objects in their entirety and to the best of our knowledge, they have not considered the tradeoff between storage and "computability over deltas". Even version control systems that provide functionality to compare multiple objects, e.g., *merge, diff, etc.*, first recreate all required files before operating upon them. Recently, [49] presented an indexing technique to support "time travel" queries for scientific arrays wherein they support *approximate* queries that can quickly identify which versions are relevant to a user and return the approximate content of these versions. However, they did not consider queries that compared the contents of two or more array versions.

Temporal indexing: There has been extensive research on temporal databases [44], and to some extent, versioned databases. Such systems have proposed various index structures to store and access historical data. For instance, Postgres used R-trees [29] to index historical data, with recent data residing in a B+Tree. More recently, Oracle 11g supports "Total Recall" feature [4] that creates read-only archives to support long-term archiving of versions. Immortal DB, which was built into Microsoft SQL Server, integrated a temporal indexing technique called the TSB-tree [37, 38] to provide high performance access and update for both current and historical data. Such systems and index structures, however, were meant for either fast complete version access or single record access to historical data. Moreover, their data structures are optimized to support a linear, temporal chain of versions.

Buneman et al. [14] proposed an archiving technique based on identifying changes to (keyed) records across versions, specifically temporal versions of hierarchical data, that are then merged into one hierarchy represented in XML format. Because they also compared against a diff-based storage solution, we present a brief comparison to highlight the respective strengths and weaknesses of the two strategies. Broadly stated, their scheme, henceforth referred to as `BA`, merges all hierarchical elements across versions into one hierarchy by identifying an element by its key and storing it only once, along with the sequence of version timestamps where the respective element appears. We reimplemented their technique in our framework, using either sorted lists or bitmaps to store the sequence of version ids where an element appears (see Appendix C for more details). Buneman et al. compared the performance of their archiver against two approaches based on deltas: (i) "cumulative diff", where every version is stored as a delta against a common (typically first) version, and (ii) "incremental diff", or "sequence-of-deltas", where every version is stored as a delta against the previous version, resulting in a line storage graph. However, cumulative diff has a large space

overhead [11, 14], and incremental diff results in large checkout times due to long delta chains.

We consider single ($k = 1$) and multiple ($k = 4$) `datafile` checkout on the dataset $\langle |\Delta| = 5\%, |A| = 1M \rangle$. Additionally, for `BA`, we vary the number of `datafiles` (N) in the archive as $N = 10, 50, 100, 250, 500, 1000$. The bitmap implementation gives superior performance (up to 19%) for $N = 100$ onwards and we use that to report checkout time, while for $N = 10, 50$, we use the sorted list implementation. As noted previously, we can pack approximately $N = 80K$ `datafiles` in a storage graph (with certain constraints) and get a delta chain of size at most 10 to checkout any single `datafile`; therefore, we set $\#\Delta = 10, 25$ for fair comparison.

	DEX; #Δ		BA; archive size (N)					
	10	25	10	50	100	250	500	1000
$k = 1$	125	550	97	388	659	1286	2677	5362
$k = 4$	263	483	144	596	1110	2484	5160	9550

Table 1: Median checkout time (ms) in DEX and `BA`

As we can see, `BA` performs better than a sequence-of-deltas approach for checkout queries. When $k = 1$ storing $N = 50$ `datafiles` gives better response times in `BA` than storing $N = 25$ `datafiles` in a sequence-of-deltas approach (demonstrated by $k = 1, \#\Delta = 25$). However, checkout times for `BA` increase rapidly as the archive size grows, and DEX is vastly superior to `BA` under more reasonable assumptions about the storage graph (in the context of a versioning/data lake scenario, it is not clear how to extend some of the optimizations in [14] that depend on the linearity of timestamps).

In short, the main difference between DEX and the sequence-of-deltas approach (that [14] primarily compared against) is that we assume that the storage graph is constructed using a technique that avoids very long delta chains (e.g., "skip links"-based approach [49], techniques that balance storage and retrieval costs [11, 34], greedy heuristic used by `git`, etc.). We further note that `BA` suffers from three major limitations: (i) the entire archive must be read even when checking out a single version, (ii) adding a new version requires an expensive merge operation that scans the entire archive (unlike a delta-oriented storage engine where only a single delta may be added), and (iii) decentralization is much more difficult in `BA` (in theory one could maintain multiple archives and merge them periodically, but we are not aware of any work that has attempted that).

Deltas and computing: The concept of making deltas "first-class citizens" was explored in Heraclitus [24]. To support "what-if" scenario analysis, they provided general-purpose constructs for creating, accessing, and combining deltas. In the specific realization of their paradigm for the relational model, deltas are a set of *signed atoms* where the positive atoms correspond to "insertions" and the negative atoms correspond to "deletions". In addition, the deltas have structure and can be manipulated directly by constructs in user programs, e.g., to delete all records satisfying a predicate. In contrast, our use of deltas is at the physical level and not exposed to the users. They do not consider optimizing the different types of queries against a delta storage. Executing queries with hypothetical state updates was also considered in [27]. Here the state updates (or deltas) were allowed to be expressions and the authors considered rewriting such queries into an optimized form based on their novel rules for substituion and the rules for relational algebra. Such rules are however not applicable in our setting. Record-based deltas were also used in [26, 51] to provide the capability of sharing data and updates among different participants. However, they focused on formalizing the semantics of the update exchange process, e.g., mapping updates across schemas and filtering them according to local trust policies, and the challenges introduced therein.

Connections to materialized views: Our techniques benefit from good storage graph constructions. Several algorithms were proposed in [11] to construct a storage graph that meets a specified set of constraints on, e.g., storage cost and retrieval cost, while also taking query workload into account. Similar problems have been considered in the context of materialized view and index selection to speed up query processing. Broadly speaking, research in this area has focused on three issues: (i) determining the search space or class of views to consider for materialization, (ii) choosing a subset of views and indexes to materialize depending on various constraints like storage overhead, maintenance overhead, effectiveness on the query workload, etc., [5, 50] and (iii) quickly determining which views to consider to answer a given query [25, 30]. In our problem setting, set-backed deltas can be considered as a form of materialized views (which can be used to reconstruct base relations), with this work addressing the problem of how to use such views to answer queries.

Evaluating set expressions: Several algorithms and data structures have been proposed in literature to solve union, intersection and difference problems on sets [7, 18, 23, 32] by minimizing the number of comparisons required. Although the ordered list representation is the most common, some algorithms also consider representing sets in other data structures, e.g., skip lists [47], machine word-based representations [21], etc., to obtain additional speedup. A comparison of few of these methods is available in [9, 19]. Speeding up set operations is largely orthogonal to our approach and we can make use of some of these techniques as additional operators with the appropriate cost model. As mentioned earlier, in this work, we use an adaptive set intersection algorithm that was shown to have reasonably good performance in [19] without requiring any preprocessing step. The larger problem here, however, is how to efficiently evaluate a set expression consisting of union, intersection and difference. [12] consider evaluating union-intersection expressions in a worst-case efficient way for a non-comparison based model. However, their approach uses hash-based dictionaries, which would require an additional pre-processing step, and it remains an open problem whether their results can be extended to handle set difference. Recently, [36] showed that, for a similar cost model, a union-intersection expression can be rewritten to perform intersections before unions with often a reduced cost. Their approach, however, did not consider rewrites in the presence of set difference.

7. CONCLUSION AND FUTURE WORK

With a growing interest in maintaining all data ever analyzed for a variety of reasons including auditing and accountability, delta-oriented storage engines (in the form of version control systems or archival systems) are becoming increasing common. Such storage engines contain a vast amount of useful information about how the datasets evolved over time, what types of analyses were performed over them, and their results; this opens up the door to executing a rich collection of introspective queries about analyses and workflows themselves. In this paper, we initiated a systematic study of this problem, by formally analyzing a simple class of queries and developing cost models and optimization algorithms for them. As we showed, even these simple queries exhibit interesting and unexplored computational challenges and the benefits of optimizing their execution can be tremendous (orders-of-magnitude in many cases). Generalizing the classes of queries and delta types are rich directions for future work that we plan to pursue towards designing a powerful, general-purpose query engine for such storage engines.

Acknowledgements: This work was supported by NSF under grants IIS-1513972 and IIS-1513443. We thank the anonymous reviewers for their valuable feedback that helped us improve the paper.

8. REFERENCES

[1] https://github.com/attic-labs/noms.

[2] https://git-lfs.github.com/.

[3] Git Packfiles. https://git-scm.com/book/en/v2/Git-Internals-Packfiles. Accessed: February 15, 2017.

[4] Oracle Total Recall with Oracle Database 11g Release 2. http://www.oracle.com/technetwork/database/focus-areas/storage/total-recall-whitepaper-171749.pdf. Accessed: November 11, 2016.

[5] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *PVLDB*, pages 496–505, 2000.

[6] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient query execution on raw data files. In *SIGMOD*, pages 241–252, 2012.

[7] R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Annual Symposium on Combinatorial Pattern Matching*, pages 400–408. Springer, 2004.

[8] J. Barbay and C. Kenyon. Deterministic algorithm for the t-threshold set problem. In *Algorithms and Computation*, pages 575–584. Springer, 2003.

[9] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM Journal of Experimental Algorithmics*, 2010.

[10] A. P. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. Elmore, S. Madden, and A. Parameswaran. DataHub: Collaborative Data Science & Dataset Version Management at Scale. In *CIDR*, 2015.

[11] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. Parameswaran. Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff. In *PVLDB*, pages 1346–1357, 2015.

[12] P. Bille, A. Pagh, and R. Pagh. Fast evaluation of union - intersection expressions. In *ISAAC*, pages 739–750, 2007.

[13] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[14] P. Buneman, S. Khanna, K. Tajima, and W. C. Tan. Archiving scientific data. *ACM Trans. Database Syst.*, 29:2–42, 2004.

[15] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 2015.

[16] A. Chavan, S. Huang, A. Deshpande, A. Elmore, S. Madden, and A. Parameswaran. Towards a Unified Query Language for Provenance and Versioning. *TaPP*, 2015.

[17] J. Cheney, S. Chong, N. Foster, M. Seltzer, and S. Vansummeren. Provenance: A future history. In *OOPSLA*, pages 957–964. ACM, 2009.

[18] E. Demaine, A. López-Ortiz, and J. Munro. Adaptive Set Intersections, Unions, and Differences. In *SODA*, pages 743–752, 2000.

[19] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, 2001.

[20] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Trans. Algorithms*, 6(1):2:1–2:19, Dec. 2009.

[21] B. Ding and A. C. König. Fast set intersection in memory. *PVLDB*, pages 255–266, 2011.

[22] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *USENIX Annual Technical Conference, General Track*, pages 113–126, 2003.

[23] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, pages 319–344, 1991.

[24] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Trans. Database Syst.*, 21(3):370–426, 1996.

[25] J. Goldstein and P.-Å. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD*, pages 331–342, 2001.

[26] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *PVLDB*, pages 675–686, 2007.

[27] T. Griffin and R. Hull. A framework for implementing hypothetical queries. In *SIGMOD*, pages 231–242, 1997.

[28] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. In *Materialized Views*, pages 145–157. 1999.

[29] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[30] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[31] P. Helland. Immutability changes everything. In *CIDR*, 2015.

[32] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1(1):31–39, 1972.

[33] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 1972.

[34] S. Khuller, B. Raghavachari, and N. Young. Balancing minimum spanning trees and shortest-path trees. *Algorithmica*, 14(4):305–321, 1995.

[35] R. K. L. Ko, P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Q. Liang, and B. S. Lee. Trustcloud: A framework for accountability and trust in cloud computing. In *IEEE World Congress on Services*, July 2011.

[36] T. Lee, J. Park, S. Lee, S. Hwang, S. Elnikety, and Y. He. Processing and optimizing main memory spatial-keyword queries. *PVLDB*, 9(3):132–143, 2015.

[37] D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal DB: Transaction time support for SQL server. In *SIGMOD*, pages 939–941, 2005.

[38] D. Lomet, M. Hong, R. Nehme, and R. Zhang. Transaction time indexing with version compression. In *PVLDB*, pages 870–881, 2008.

[39] J. MacDonald. File system support for delta compression. 2000.

[40] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. G. Parameswaran, and A. Deshpande. Decibel: The relational dataset branching system. *PVLDB*, 9(9):624–635, 2016.

[41] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *SIGCOMM*, pages 181–194, 1997.

[42] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *SIGMOD*, pages 100–111, 1997.

[43] E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, pages 251–266, 1986.

[44] G. Özsoyoğlu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. on Knowl. and Data Eng.*, pages 513–532, 1995.

[45] J. Paulo and J. Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys*, 2014.

[46] S. Rönnau, J. Scheffczyk, and U. M. Borghoff. Towards XML Version Control of Office Documents. In *Proc. of the ACM Symposium on Document Engineering*, pages 10–19, 2005.

[47] P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 71–83, 2007.

[48] A. Seering, P. Cudré-Mauroux, S. Madden, and M. Stonebraker. Efficient versioning for scientific array databases. In *ICDE*, pages 1013–1024, 2012.

[49] E. Soroush and M. Balazinska. Time travel in a scientific array database. In *ICDE*, pages 98–109, 2013.

[50] Z. A. Talebi, R. Chirkova, Y. Fathi, and M. Stallmann. Exact and inexact methods for selecting views and indexes for OLAP performance improvement. In *EDBT*, pages 311–322, 2008.

[51] N. E. Taylor and Z. G. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD*, pages 13–24, 2006.

[52] D. J. Weitzner, H. Abelson, T. Berners-Lee, J. Feigenbaum, J. Hendler, and G. J. Sussman. Information accountability. *Communications of the ACM*, 51(6):82–87, 2008.

APPENDIX

A. ALGORITHMS

We outline the pseudocode for the missing algorithms below.

A.1 Path Contraction (PC)

The input to PC is a materialized `datafile`, say M, followed by a sequence of deltas, say $\Delta_1, \ldots, \Delta_m$ and the output is the `datafile` represented by the corresponding delta expression, $M \oplus \Delta_1 \oplus \cdots \oplus \Delta_m$. The algorithm uses the property that if the optimal solution splits the contraction of a path of length m into two sub-paths, then the contraction of each of the two sub-paths must be optimal (otherwise, we can improve the solution for the sub-paths to enhance the overall solution). For $0 \le i \le j \le m$, let $C[i, j]$ denote the minimum cost to contract the sequence $\Delta_i, \ldots, \Delta_j$, $D[i, j]$ denote the estimated size of the corresponding intermediate result, and $S[i, j]$ denote how to best split the sequence. For notational convenience only, if M is present, $\Delta_0 = M$. The pseudocode of PC is shown in Algorithm 1.

A.2 Tree Contraction (TC)

The input to TC is an access tree $\mathcal{G}_Q$ and the output is the set of `datafiles` $\mathcal{A}_k$. The pseudocode of TC is shown in Algorithm 2 and we explain it with the help of Fig. 14.

First, note that if $\mathcal{G}_Q$ has more than one materialized `datafile`, then we can consider the sub-trees rooted at each materialized file independently. Fig. 14(a) shows an access tree to checkout A_1, A_2, A_3. Let M be the root of this access tree. Starting from M, let B_i be the first node that has $l > 1$ children. Let $B_0, \ldots, B_{i-1}$ be the intermediate nodes on the $M - B_i$ path. Let $\mathcal{G}_Q(B_j)$, $0 \le j < i$, be the access tree rooted at B_j (this tree is equivalent to deleting the nodes $M, B_0, \ldots, B_{j-1}$ from $\mathcal{G}_Q$). For example, Fig. 14(b) shows two components: (i) the path $\rho(B_{i-1})$ (above), and (ii) the access tree $\mathcal{G}_Q(B_{i-1})$ (below). Let `split`$(\mathcal{G}_Q)$ denote the operation that splits $\mathcal{G}_Q$ at B_i into l access trees, one for each child of B_i. This is shown in Fig. 14(c). Let `split-par`$(\mathcal{G}_Q)$ denote the operation that splits $\mathcal{G}_Q$ at B_i into l access trees, one for each child of B_i, but this time preserving the parent sequence of deltas in each split. This is shown in Fig. 14(d).

Algorithm 1: Path Contraction (PC)

Input : A materialized `datafile` M (optional); $\Delta_1, \ldots, \Delta_m$
Result: Minimum cost to evaluate $M \oplus \Delta_1 \oplus \ldots \Delta_m$

for $l \leftarrow 2$ *to* m **do**
  for $i \leftarrow 0$ *to* $m - l + 1$ **do**
    $j \leftarrow i + l - 1$
    $C[i, j] \leftarrow$ $\min_{i \le k < j} C[i, k] + C[k+1, j] + C_{\oplus}(D[i, k], D[k+1, j])$
    $S[i, j] \leftarrow \arg\min_{i \le k < j} C[i, k] + C[k+1, j] + C_{\oplus}(D[i, k], D[k+1, j])$
    `/* Update estimated size of` $\Delta_i \oplus \cdots \oplus \Delta_j$ `*/`
    $D[i, j] \leftarrow$ `EstimateSize`$(D[i, S[i, j]], D[S[i, j] + 1, j])$
  end
end
return $C[1, m]$ *(final cost) and* S *(splitting markers)*

Algorithm 2: Tree Contraction

Input : Access Tree $\mathcal{G}_Q$
Apply PC for each $\rho(A_i)$, $A_i \in \mathcal{A}_k$. Memoize $C[]$, $S[]$ and $D[]$.
return `Best-Subexp(`$\{\}, \mathcal{G}_Q$`)`

Procedure `Best-Subexp(`$\mathcal{L}, \mathcal{G}_Q$`)`
  Input : Deltas, $\mathcal{L}$; Access tree, $\mathcal{G}_Q$
  if $\mathcal{G}_Q$ is a path **then return** Best solution for $\{\mathcal{L} \cup \mathcal{G}_Q\}$
  forall $0 \le j < i$ **do**
    $\Delta_{\rho(B_j)} \leftarrow$ estimated delta for the sequence $\rho(B_j)$
    $\mathcal{L}' = \mathcal{L} \cup \Delta_{\rho(B_j)}$
    cost_g $\leftarrow$ `Best-Subexp(`$\mathcal{L}', \mathcal{G}_Q(B_j)$`)`
  $\Delta_{\rho(B_i)} \leftarrow$ estimated delta for the sequence $\rho(B_i)$
  $\mathcal{L}' = \mathcal{L} \cup \Delta_{\rho(B_i)}$
  cost_g $\leftarrow \sum_{\mathcal{G}'_Q \in$ `split(`$\mathcal{G}_Q$`)` $}$ `Best-Subexp(`$\mathcal{L}', \mathcal{G}'_Q$`)`
cost_g $\leftarrow \sum_{\mathcal{G}'_Q \in$ `split-par(`$\mathcal{G}_Q$`)` $}$ `Best-Subexp(`$\mathcal{L}, \mathcal{G}'_Q$`)`
return *Best cost_g*

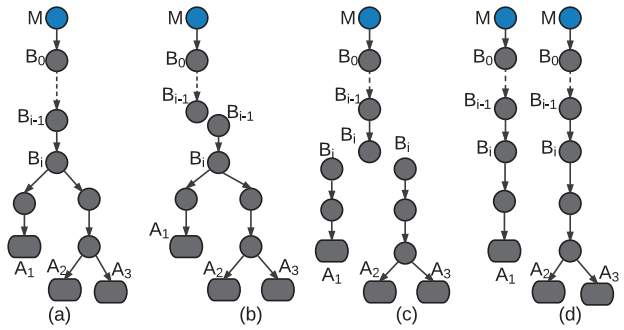


Figure 14: An instance of the Tree Contraction algorithm; (a) is an access tree for the query Checkout(A_1, A_2, A_3).

At a high level, TC breaks up the problem into two questions: how do we decide which sub-expressions to share and how do we best parenthesize each (sub-)expression? We already know how to compute the solution for the latter using PC. Therefore, we begin with

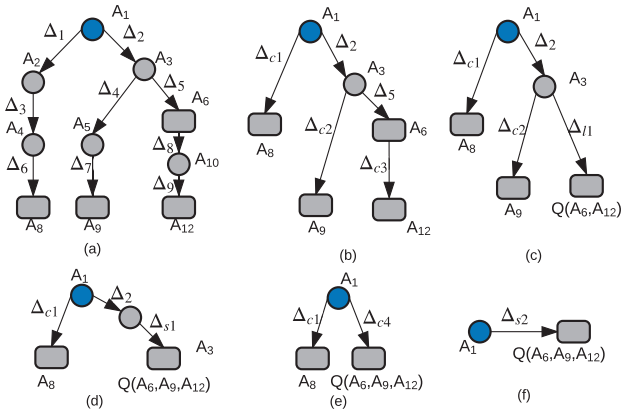


Figure 15: Access tree during the progress of C&R

applying PC for all $\rho(A_i), A_i \in \mathcal{A}_k$. However, apart from the best solution for the complete expression, we also return the following information about each path $\rho(A_i)$, all of which is computed by PC during its execution. Let $\Delta_1, \ldots, \Delta_m$ be the sequence of deltas on the path $\rho(A_i)$ and i, j be indices such that $0 \leq i \leq j \leq m$. Then we return: (i) the minimum cost to contract the sequence $\Delta_i, \ldots, \Delta_j$, denoted by $C[i, j]$, (ii) the estimated size of the intermediate delta, denoted by $D[i, j]$, and (iii) the split marker, indicating how to best split the the sequence, denoted by $S[i, j]$. We re-use these estimates of intermediate delta sizes and partial contraction costs wherever the corresponding sub-expressions are considered for sharing. To decide which sub-expression to share, we simply enumerate all possibilities for the sub-expression starting from the root of the access tree (this is done in line 5). The rest of the problem is solved recursively where $\mathcal{L}$ is the sequence of sub-expressions that are considered to be shared. Finally, we also need to account for the possibility of not sharing any sub-expression.

The time complexity of TC is $O(m^3)$ where m is the number of deltas in the access tree.

A.3 Contract and Reduce (C&R)

Algorithm 3 describes the general form of C&R for all queries. Specifically, it takes as input a query $Q \in \{I, U, T_t\}$, and its access tree $\mathcal{G}_Q$, and outputs the appropriate result. We describe its behavior with the help of the following example.

Algorithm 3: Contract and Reduce (C&R)

Data: Query $Q \in \{I, U, T_t\}$ and access tree $\mathcal{G}_Q$
1 **while** $\mathcal{G}_Q$ *contains more than one delta* **do**
2 **Contract Phase:**
3 $P \leftarrow$ list of *maximal continuous* delta paths of length > 2 in $\mathcal{G}_Q$
4 Contract all paths in P using PC and update $\mathcal{G}_Q$
5 **if** $\mathcal{G}_Q$ *becomes a line/star* **then**
6 **return** $R \leftarrow$ *result using* **H1**
7 **Reduce Phase:**
8 $L \leftarrow$ list of *line* structures in $\mathcal{G}_Q$
9 Reduce each line in L based on Q, i.e., apply one of **T1/T3**, and update $\mathcal{G}_Q$
10 $S \leftarrow$ list of *star* structures in $\mathcal{G}_Q$
11 Reduce each star in S based on Q, i.e., apply one of **T2/T4/T5**, and update $\mathcal{G}_Q$
/* At this point $\mathcal{G}_Q$ contains one delta. */
12 **return** $R \leftarrow Root(\mathcal{G}_Q) \oplus \Delta$

Example 15. *Consider the query $I(A_6, A_8, A_9, A_{12})$ with access tree as in Fig. 15(a). In the first iteration, during the contract phase, we compute the deltas: (1) $\Delta_{c1} = \Delta_1 \oplus \Delta_3 \oplus \Delta_6$, (2) $\Delta_{c2} = \Delta_4 \oplus \Delta_7$, and (3) $\Delta_{c3} = \Delta_8 \oplus \Delta_9$ (Fig. 15(b)). In the reduce phase where we reduce A_6 and A_{12} arranged in a line (Fig. 15(c)). This reduction puts A_9 and $Q(A_6, A_{12})$ in a star, which is then reduced as in Fig. 15(d)). In the next iteration, during the contract phase, we compute $\Delta_{c4} = \Delta_2 \oplus \Delta_{s1}$ (Fig. 15(e))). In the reduce phase, we reduce the star A_8 and $Q(A_6, A_9, A_{12})$ which leaves the access tree with a single delta.*

A.4 Multiset Delta Contraction

During the execution of C&R for t-threshold queries, we will encounter expressions of the form $\Delta_x \oplus \Delta_y$, where Δ_y is a multiset-based delta. Here, we outline a linear time algorithm (shown in Algorithm 4) that computes the contraction in a manner that helps us compute the query result. Recall that $p(\Delta)$ is the number of datafiles in $\mathcal{A}_k$ that are reduced by Δ and c is the multiplicity of a record.

Algorithm 4: Patch operation for multiset-based deltas

Data: Set-backed delta Δ_x, and multiset-based delta Δ_y
Result: Multiset delta $\Delta = \Delta_x \oplus \Delta_y$
1 Initialize $\Delta \leftarrow \Delta_y, p \leftarrow p(\Delta) \leftarrow p(\Delta_y)$
2 **for** $r \in \Delta_x^+$ **do**
3 **if** $r \in \Delta^-$ **then**
4 Remove (r, c) from Δ^-, Add $(r, p - c)$ to Δ^+
5 **else**
6 Add (r, p) to Δ^+
7 **for** $r \in \Delta_x^-$ **do**
8 **if** $r \in \Delta^+$ **then**
9 Remove (r, c) from Δ^+, Add $(r, p - c)$ to Δ^-
10 **else**
11 Add (r, p) to Δ^-
12 **return** Δ

B. DATASET GENERATION

Lacking access to real-world versioned datasets with sufficient and varied structure, we instead developed a synthetic data generator to generate datasets with very different characteristics for a wide variety of parameter values. This enables us to carefully study the performance of our techniques in various settings. Formally, every dataset is characterized by a 4-tuple, $\langle T, |A|, |\Delta|, \#\Delta \rangle$, where the meaning of each paramter is as follows.

- **Structure of access tree (T):** Since Algorithm 3 makes decisions based on the presence of lines and stars, we control their occurrence when selecting datasets. Figure 16 shows three different types of access trees that we consider during evaluation: *line-shaped* (l), *star-shaped* (s) and *line-and-star* (ls).
- **Average datafile size ($|A|$):** This controls the average number of records in a datafile. Each record is a tuple consisting of a random 64 byte string, integer and double values.
- **Average delta size ($|\Delta|$):** This controls the average size of the deltas in the dataset. We represent this in terms of percent size compared to $|A|$.
- **Number of deltas in access tree ($\#\Delta$):** This parameter controls the number of number of deltas in the access tree and hence the overall deltas that are considered during execution.

Parameter	Explanation	Values
k	Query size	2, 4, 6, 8, 10
$\|A\|$	Average `datafile` size	1 million(M), 2M, 3M, 4M, 5M
$\|\Delta\|$	Average delta size	1%, 2%, 3%, 4%, 5%
$\#\Delta$	Number of deltas	10, 25, 50, 75, 100
T	Shape of access tree	Line (l), Star (s), Line-and-star (ls)

Table 2: Possible values of parameters characterizing a synthetic dataset

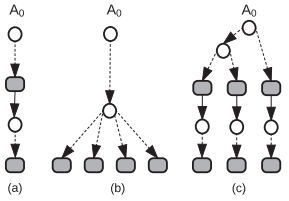


Figure 16: Access tree shapes; (a) Line, (b) Star, (c) Line-and-star

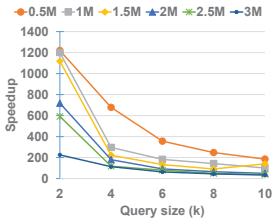


Figure 17: Effect of bitmap size

C. COMPARISONS WITH TEMPORAL INDEXING [14]

In this section, we present further background on the temporal indexing techniques by Buneman et al. [14], and discuss how we reimplemented and compared against their approach.

Broadly speaking, their technique, referred to as BA henceforth, focuses on storing multiple temporal versions of a hierarchical dataset (e.g., a large XML document). At a high level, BA merges all hierarchical elements across the versions into a single hierarchy by identifying an element by its key and storing it only once, along with the sequence of version timestamps (as intervals) where the respective element appears. Answering a checkout query thus requires scanning the entire archive, and using the intervals to decide which elements belong to the answer.

We re-implemented the BA in-memory archiving algorithm in our framework as faithfully as we could. We represent a dataset of records as a one-level hierarchical document with all the records as children of the root node. When merging two datasets into a single archive, we identify the common records and only store them once. Due to the nonlinear nature of "version ids" (unlike timestamps) in our problem setting, we tried two different implementations to keep the set of version ids for an element/record: (1) a sorted list or (2) a bitmap. In the sorted list implementation, the version ids associated with every record are stored in a sorted array and during retrieval, we use binary search to decide if the record is present in the desired version. In the bitmap implementation, a bitmap of size equal to the number of versions in the archive is used with each record to indicate the versions that the record is present in. We use the roaring bitmap library [15] to store these bitmaps. During retrieval, a simple scan through the archive can retrieve any version. We note that it is not clear how to extend some of the optimizations in Buneman et al., most notably "timestamp trees", that depend on the linearity of timestamps, to the nonlinear nature of version ids in a decentralized versioning/data lake scenario.

D. EXPERIMENTS WITH BITMAP DELTAS

We have also built support for a filtered index to answer intersection and union queries, and we show the results for an illustrative experiment. Akin to a relational database, a filtered index in DEX is suited to answer queries that always select from a finite "universal" set of records. In this case, we can encode a set of records using a bitmap, where the order of records is determined by their SHA1 value. The index creation step creates a bitmap of size $|\mathcal{K}|$ for each materialized `datafile` and two bitmaps for each delta in the storage graph. We can then use the bitwise AND($\wedge$), OR($\vee$) and NOT($\neg$) operations to compute set intersection, union and difference. In this experiment, we use a compressed bitmap library called *roaring bitmaps* [15] . Fig. 17 shows the effect of index size on the intersect query. Here, we measure the speedup vs query size for index size ranging from 500K-3M records. As expected, for small universal sets, we get largely improved speedup ratios (up to **1200X**). With large universe sizes, there is however a penalty incurred when selecting the records themselves given the bitmap information.