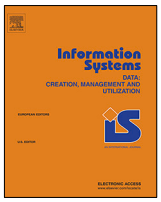
Information Systems 113 (2023) 102144

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is



Multi-valued indexing in Apache AsterixDB (SI DOLAP 2022)

Glenn Galvizo *, Michael J. Carey

Department of Computer Science,University of California, Irvine, 92697, California, USA

ARTICLE INFO

Article history:
Received 17 June 2022
Received in revised form 14 October 2022
Accepted 31 October 2022
Available online 10 November 2022
Recommended by Dennis Shasha

Keywords:
Multi-valued indexing
Index specification
Index implementation
Query optimization
AsterixDB

ABSTRACT

Secondary indexes in relational database systems are traditionally built under the assumption that one data record maps to one indexed value. Nowadays, particularly in NoSQL systems, single data records can hold collections of values that users want to access efficiently in an ad-hoc manner. Multi-valued indexes aim to give users the best of both worlds: (i) to keep a more natural data model of records with collections of values, and (ii) to reap the benefits of a secondary index.

In this paper, we detail the steps taken to realize multi-valued indexes in AsterixDB, a Big Data management system with a structured query language operating over a collection of documents. This includes (a) creating the specification language for such indexes, (b) illustrating data flows for bulk-loading and maintaining an index, and (c) discussing query plans to take advantage of multi-valued indexes for use in predicates with existential and universal quantification. We conclude with experiments that measure the impact of maintaining an AsterixDB multi-valued index and experiments that compare the query performance our multi-valued indexes against similar indexes in MongoDB and Couchbase Server's Query Service.

1. Introduction

Multi-valued fields, such as arrays and multisets, are a staple in many (if not all) NoSQL systems. Secondary indexes are traditionally for *single-valued* fields, where a record in a database maps to one entry in an index (e.g. a leaf node in a B+ tree index). Here, we will refer to a secondary index on a single-valued field for a collection of records as a single-field single-valued index, while a secondary index over multiple single-valued fields will be referred to as a *composite* single-valued index. A multi-valued index is a secondary index on a multi-valued field. A multi-valued index is distinct from a single-valued index, as the number of values associated with the multi-valued field is not known a priori.

Given a collection of records to index, this work focuses on supporting secondary indexes for multi-valued fields in *Apache AsterixDB*. Apache AsterixDB is a NoSQL-style Big Data management system with a declarative query language (SQL++), a rule-based query optimizer, a parallel dataflow execution engine, and partitioned LSM-based storage and indexing. The main contributions of this paper are as follows:

1. An approach that separates the implementation of multi-valued indexes from the low-level storage layer of a database, yielding a clean architecture with the additional benefit of being able to accommodate index structures other than B+ trees. We address the challenges of (a) completeness (i.e. what information must be stored in a multi-valued index) and (b) uniqueness (i.e. how should duplicate items in a multi-valued field be handled).
2. A multi-valued index specification language. We address the challenge of defining a language that is neither ambiguous with respect to structure nor verbose, motivated by the absence of an existing language that satisfies both properties.
3. Foundations for bulk loading and maintaining multi-valued indexes. We address the challenge of designing efficient loading and maintenance strategies that must be transactionally compliant.
4. Details on query evaluation for two types of queries involving arrays and multisets: existential quantification and universal quantification. This includes join queries that probe items in another dataset's multi-valued field. We address two main challenges here: (a) building query plans that are transactionally compliant, and (b) finding a mapping between the structure of a query and an applicable multi-valued index, if any exist.
5. Three sets of experiments: (a) one that measures the impact of maintaining multi-valued indexes in AsterixDB, (b) one that evaluates the efficacy of such indexes for applicable queries, and (c) and one that measures how our implementation fares against those in two other document databases: MongoDB and the Couchbase Query Service.

* Corresponding author.
E-mail addresses: ggalvizo@uci.edu (G. Galvizo), mjcarey@uci.edu (M.J. Carey).

https://doi.org/10.1016/j.is.2022.102144

The rest of this paper is structured as follows: Section 2 details related work around multi-valued indexing. Section 3 reviews Apache AsterixDB, the big data management system used for this research. Section 4 describes the syntax for specifying multi-valued index creation statements. Section 5 discusses various data flows to realize multi-valued indexing. Section 6 evaluates the maintenance impact and performance of such indexes. Section 7 concludes the paper and details potential future work with respect to multi-valued indexing.

2. Related work

The advent of nesting in data models for databases beyond the flat relational era has brought with it a set of challenges with respect to associative access. Related work can be grouped into two general areas: (i) indexing in object-oriented databases, and (ii) multi-valued indexing in modern document databases (document stores, key–value stores with document extensions, and relational stores with document extensions).

2.1. Indexing in object-oriented databases

We start our discussion with the object data model, with work in this area dating back approximately 30 years. Here, objects *and* their member objects are each first class citizens. In terms of indexing, object-oriented databases must address the problem of what exactly one should index when objects can reside in objects.

Fig. 1 depicts three classes: `Vehicle`, `Manufacturer`, and `Division`. A `Vehicle` is produced by a single `Manufacturer`, and a `Manufacturer` can possess one or more `Division` instances. Suppose we want to index vehicles by the names of their vehicle manufacturers. More specifically want to index the `ManName` attribute inside the `Manufacturer` object of a `Vehicle` object. Bertino and Kim studied three approaches to nested indexes in object databases [1]: (i) nested indexes (which map `ManName` attribute values to the `Vehicle` objects), (ii) path indexes (which map `ManName` attribute values to both `Manufacturer` and `Vehicle` objects), and (iii) multi-indexes (which first map the `ManName` attribute value to the `Manufacturer` objects, then map the `Manufacturer` objects to the `Vehicle` objects). Under a relational lens, multi-indexes (item 2.1) can be viewed as pair-wise join indexes, which have also been studied by Valduriez [2]. Bertino and Foscoli address the problem of incorporating the notion of inheritance (e.g. `Moped`, a child class of `Vehicle`) with indexing nested objects [3]. Kemper and Moerkotte detail an approach based on indexing objects that are nested in sets and lists [4]. As an example, suppose we now want to index all `DivName` field values associated with all `Division` objects within the `Divisions` list of a `Manufacturer` object. Indexing `DivName` field values here is multi-valued indexing with an object-oriented twist, and support for such indexes can be found in many of the object databases of this era [5–8]. Goczyla proposed an extension to set indexing in object databases that not only handles set membership, but also the more general cases of superset, subset, and set equality [9].

2.2. Multi-valued indexing in document databases

Next we address the document model, where documents themselves are self-describing (lending the model to weaker type assumptions). Consider the XML document model, where an element is composed of many sub-elements and there exists no way to determine if a sub-element will be single-valued or multi-valued. The XML extension for DB2 addresses the single-valued vs. multi-valued problem with respect to indexing by treating every element as a potential multi-valued attribute [10]. The JSON document model, in contrast to the XML document model, does allow one to specify if a field is multi-valued or not (making the single-valued vs. multi-valued problem a non-issue). Modern JSON document stores such as Couchbase [11], MongoDB [12], and Oracle's NoSQL database [13] have support for multi-valued indexing, but all had somewhat different design goals than the multi-valued indexing approach studied here. The array indexes of the Couchbase Index Service were designed with the intent to fully *cover* certain queries (i.e., to use only the index to satisfy a query), while AsterixDB's multi-valued indexes are designed to handle a larger set of queries at the cost of no longer being covering. The Couchbase Index Service does also offer non-covering multi-valued "Flex Indexes" [14], made with the intent to handle a larger set of queries that can be answered using an inverted index. In contrast, multi-valued indexes in AsterixDB were designed to support the kinds of queries that can be answered using a B+ tree. Finally, MongoDB's and Oracle's index specification syntax leave ambiguities for the user (in terms of structure and needless repetition, as we will discuss later).

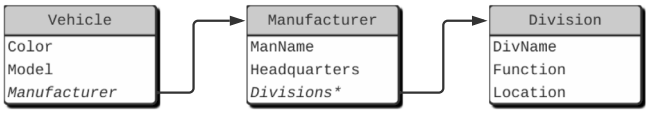


Fig. 1. Object-oriented schema for an example vehicle-manufacturer.

The document model is not exclusive to document databases; the model has also found adoption in several key–value stores and modern relational systems. ArangoDB and CockroachDB offer array indexes, but only to satisfy membership queries (i.e. no range predicates) on non-nested arrays [15,16]. Relational databases with document extensions like MySQL [17] and PostgreSQL [18] also support a limited form of multi-valued indexing, but again only support membership queries. The multi-valued indexes in AsterixDB, on the other hand, are designed to support a much larger set of queries, such as joins with a value inside a multi-valued field, existential quantification, and universal quantification.

3. Background

In this section we give an overview of Apache AsterixDB, several example AsterixDB datasets, and single-valued B+ tree indexes in AsterixDB.

3.1. AsterixDB system overview

AsterixDB is a big data management system (BDMS) designed to be a highly scalable platform for information storage, search, and analytics [19]. To scale outward it follows a shared-nothing architecture, where each node independently accesses storage and memory. Fig. 2 depicts the software stack, consisting of AsterixDB and two data model independent layers (Algebricks and Hyracks, also used for projects other than AsterixDB [20]). All nodes in an AsterixDB cluster are managed by a central cluster controller that both serves as an entry point for user requests and coordinates work amongst the individual AsterixDB nodes. After a request arrives at the cluster controller, the request is first translated into a logical plan and subsequently given to a rule-based optimizer (i.e. Algebricks) to produce an optimized logical plan [21]. This optimized logical plan is then translated into a job that is distributed by the Hyracks runtime engine and executed across all nodes in the cluster [22]. Datasets in AsterixDB are partitioned across the cluster on their primary key into primary B+ tree indexes, where the data records reside, with all secondary indexes being local to each node. Natively, all datasets and indexes in AsterixDB use LSM (log-structured merge) trees to efficiently ingest new data [23].

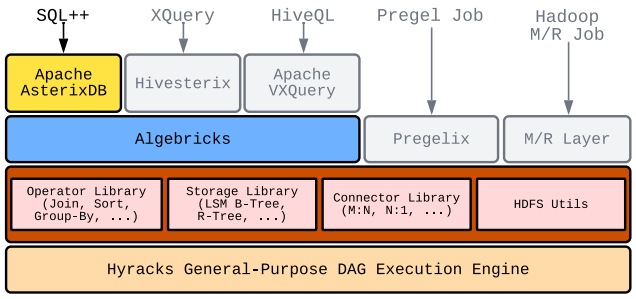


Fig. 2. Overview of the Asterix software stack. The components of interest are Apache AsterixDB, Algebricks, and Hyracks.

3.2. Inventory management example

To illustrate the topics mentioned in the following sections, an inventory management example will be used. There are three datasets associated with this example: (i) `Users`, who use (ii) `Stores` to purchase products using (iii) `Orders`.

3.2.1. Users dataset

The `Users` dataset represents customers of a shopping service who want to place orders. A user is uniquely identified by their `user_id`, having an optional `email` field, having a `name` composed of a `first` and `last` part, and having zero or more `phones` (each phone being composed of a `kind` and a `number`).

```
CREATE TYPE UsersType AS {
    user_id: string,
    email: string?,
    name: { first: string,
            last: string },
    phones: [{ kind: string,
               number : string }] };
```

Listing 1: Type definition for the `Users` dataset.

3.2.2. Stores dataset

The `Stores` dataset represents stores that sell products to users through orders. A store is uniquely identified by a `store_id` and contains a `name`, an `address` (composed of a `street`, `city`, `state`, and a `zip_code`), and a list of categories describing what the store sells (`categories`).

```
CREATE TYPE StoresType AS {
    store_id: string,
    name: string,
    address: { street: string,
               city: string,
               state: string,
               zip_code: string },
    categories: [string] };
```

Listing 2: Type definition for the `Stores` dataset.

3.2.3. Orders dataset

The `Orders` dataset represents orders placed by users to some store. An order is uniquely identified by an `order_id` and has one-to-many relationships with `Users` and `Stores` (represented as `user_id` and `store_id`). Each order has a list of line items, with each line item being uniquely identified by its `item_id` (with respect to the containing order itself), a `qty`, a one-to-many relationship with an unlisted dataset `Products` (represented as `product_id`), and an array of user-specified `tags`.

```
CREATE TYPE OrdersType AS {
    order_id: string,
    user_id: string,
    store_id: string,
    items: [{ item_id: string,
              qty: integer,
              product_id: string,
              tags: [string] }] };
```

Listing 3: Type definition for the `Orders` dataset.

3.3. Single-Valued B+ tree indexing

AsterixDB provides a choice of several secondary index types to accelerate queries: `BTREE` (the default), `RTREE`, `KEYWORD`, `NGRAM`, and `FULLTEXT`. A `CREATE INDEX` statement in AsterixDB consists of three main parts: (1) the name of the index, (2) the dataset to build the index on, and (3) an ordered list of *paths* to fields of the dataset to index. A path is a dot-separated list of fields, where a dot denotes that the field after the dot can be found inside the object field before the dot. The complete syntax for a `CREATE INDEX` statement is given in Fig. 3.

Suppose that we want to create a composite secondary B+ tree index for the `Users` dataset on two fields: the `name` field and the `zip_code` field inside of an object field `address`. To create our desired index, we would issue the statement in Listing 4. The `name` field is not nested inside any object, so we simply use `name` in our index creation statement. `zip_code` however is nested inside the `address` field, so we use the path `address.zip_code`. The use of the dot to express nested fields here generalizes to all types of single-valued (i.e. no arrays or multisets) object nesting structures.

```
CREATE INDEX userNameZipIdx ON Users (
    name, address.zip_code
);
```

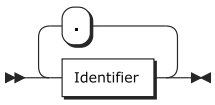
Listing 4: Example specification for a single-valued composite secondary B+ tree index in AsterixDB.

Once an AsterixDB user issues the index creation statement in Listing 4, users can expect queries that quantify over the `name` field or the `name` and `zip_code` fields from the `Users` dataset to utilize the index in their evaluation.

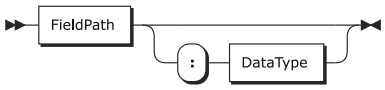
4. Multi-Valued index specification

We itemize the requirements for a user-friendly multi-valued index specification below:

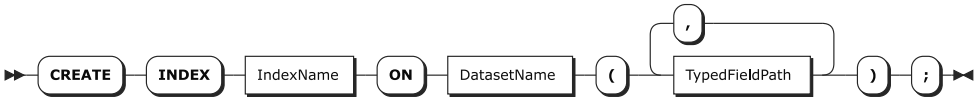
1. Distinguish between single-valued and multi-valued fields. Similar to single-valued indexes, users must be able to describe the type and structure of the fields they want to be indexed.
2. Allow fields within the same array/multiset field to be indexed, but not fields that span across different multi-valued fields. Consequently, we should abstain from specifying a multi-valued field more than once to improve legibility.
3. Constrain the specification language. We are not interested in creating multi-valued indexes that act as the sole data source for a few queries (i.e. covering indexes), so we should not complicate the syntax by allowing general expressions to be indexed.



(a) Syntax diagram for specifying an index field (`FieldPath`).

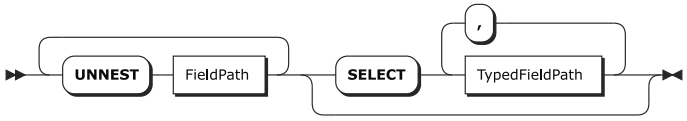


(b) Syntax diagram for a typed version of `FieldPath` (`TypedFieldPath`).

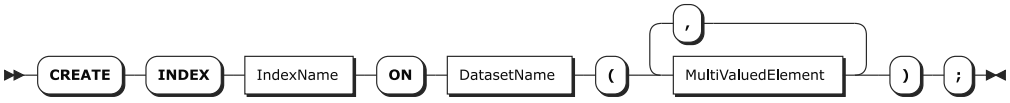


(c) General syntax diagram for a `CREATE INDEX` statement.

Fig. 3. DDL syntax for creating a single-valued index in AsterixDB.



(a) Syntax diagram for a multi-valued element.



(b) General syntax diagram for a `CREATE INDEX` statement involving multi-valued elements.

Fig. 4. DDL syntax for creating a multi-valued index in AsterixDB.

```
1  CREATE INDEX storesCatIdx ON Stores (
2      UNNEST categories
3  );
4
5  CREATE INDEX ordersItemIDIdx ON Orders (
6      UNNEST orderline
7      SELECT item_id
8  );
9
10 CREATE INDEX ordersTagIdx ON Orders (
11     UNNEST orderline
12     UNNEST tags
13 );
```

Listing 5: Example specification for three multi-valued indexes.

```
1  CREATE INDEX userNumberKindIdx ON Users (
2      UNNEST phones
3      SELECT number, kind
4  );
5
6  CREATE INDEX userNameNumberIdx ON Users (
7      name,
8      UNNEST phones
9      SELECT number
10 );
```

Listing 6: Example specification for two composite multi-valued indexes.

4. Have an easy-to-read index specification. Ideally, users should be able "debug" their index specification by issuing a query that closely follows their index specification itself.

Our solution is to introduce two keywords into the `CREATE INDEX` statement, borrowed from the AsterixDB query language: `UNNEST` and `SELECT`. Users can then specify a *multi-valued element* in lieu of a field or field path inside the existing `CREATE INDEX` grammar. A multi-valued element starts with a series of `UNNEST` terms, which describe the nesting structure of multi-valued field(s). If the desired field to index is located within an array or multiset of *objects*, then '`SELECT`' followed by the desired field/field path is specified. Lastly, if the type of the field is not specified with the dataset DDL, then a user concludes with the type name. The syntax for a multi-valued element is given in Fig. 4.

We demonstrate several examples in Listing 5 for the datasets described in Section 3.2, some of which will also serve to guide discussion in the following section. The first statement in Listing 5 indexes the categories associated with a store, where a category is within a multi-valued field. The second statement creates an index on the item IDs within the orderlines of an order. This statement demonstrates the use of '`SELECT`' to specify fields of an object inside of a multi-valued field. The third statement creates an index on values inside of a multi-valued field of an object that itself is located within a multi-valued field. Here, we exhibit the use of multiple `UNNEST` terms to identify deeper multi-valued nested structures.

The two statements in Listing 6 specify composite indexes that involve a multi-valued element. The first statement creates a composite multi-valued index on two fields within an array of objects. The first statement shows how our index specification syntax avoids repeating the nesting structure (in this case, the `phones` array) for multiple values inside the same array. The second statement creates an index on the user name and phone numbers associated with a given user, where a given phone number is contained in an object within a multi-valued field. Note that `userNameNumberIdx` is a composite multi-valued index where a single-valued field and a multi-valued field coexist in the same index. Its statement also illustrates the benefits of not altering the rest of the `CREATE INDEX` grammar: the specification for composite indexes containing both single-valued fields and

```
{ "store_id": "A34AD",
  "name": "Raspberry Store",
  "categories": ["Produce",  "Snacks"] }
{ "store_id": "1939D",
  "name": "Raspberry Store",
  "categories": ["Hardware", "Hardware"] }
```

Listing 7: Two sample documents from the Stores dataset.

multi-valued fields is nearly identical to the specification for composite single-valued indexes.

5. Index implementation

The implementation of multi-valued indexes can be broken into four main sections: (i) what an index entry is, (ii) how we bulk load an index, (iii) how we maintain an index, and (iv) how we utilize indexes in queries.

5.1. Defining an index entry

We will describe the index entries for a multi-valued index of type BTREE, but it is important to stress that this work is general enough to also be applied to RTREE indexes in the future. A leaf node in a B+ tree must minimally contain two items: (i) the field value(s) that the tree is sorted on (i.e. the values of the sort key), and (ii) the associated payload (i.e. the data record(s) or way(s) to get to the data record(s)). For AsterixDB, item (ii) is a singular unique field: the primary key associated with the record being indexed. A total order on B+ trees in the presence of potentially duplicate index field values is maintained by adding the record's primary key as a suffix to the sort key itself. Suppose that a single-valued index on the name field of the Stores dataset were built. Given the two documents in Listing 7, the two resulting index keys for this single-valued index would be <"Raspberry Store", "A34AD" and <"Raspberry Store", "1939D">.

Leveraging the data model independence offered by both Algebricks and Hyracks, an index entry in a multi-valued B+ tree index is really no different physically than an index entry in an single-valued B+ tree index. Multi-valued indexes are thus able to work above the low-level storage layer in AsterixDB. For a multi-valued field of an index, the sort key is drawn from the values inside the multi-valued field (not the enclosing field value itself). Using the index on the categories string array of the Stores dataset as an example, we differentiate between the individual items of the categories array (e.g. "Produce", "Snacks", etc…) and the enclosing multi-valued field itself, categories.

The approach of creating keys from values inside of a multi-valued index introduces a new issue: how do we handle duplicate values in a multi-valued field? A given primary key appears at most once in an single-valued index. This is no longer true with multi-valued indexes, as a primary key value can now be associated with multiple index entries. We demonstrate this with the resulting sort key + primary key pairs of the storesCatIdx index for the documents in Listing 7: <"Produce", "A34AD">, <"Snacks", "A34AD">, and *two* instances of <"Hardware", "1939D">. This non-uniqueness leads to several issues, the most notable being concurrency (discussed in Section 5.3.1). Given that multi-valued covering indexes are not in the scope of this research, the question arises: "Is it even necessary to store duplicate values for a single record's multi-valued field?" (e.g. the two instances of the <"Hardware", "1939D"> above). Existential quantification queries and universal quantification queries can be answered without the inclusion of these duplicate keys, so the decision was made to simply not store more than one distinct value per record's multi-valued field in the first place.

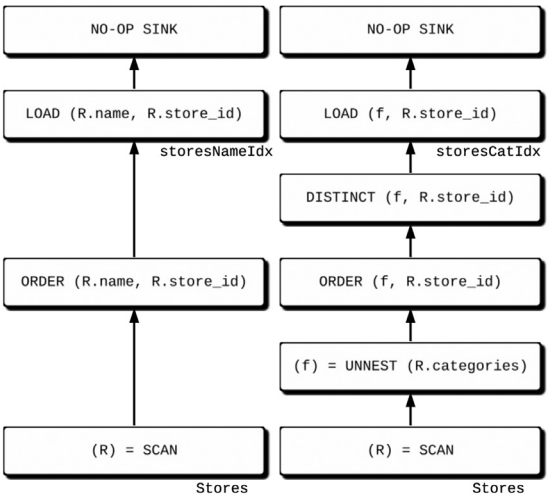


Fig. 5. Two separate data flows for bulk-loading an index. The left one is the data flow for bulk-loading a single-valued index, while the right one is the data flow for bulk-loading a multi-valued index.

5.2. Bulk loading an index

In AsterixDB, there are two cases where bulk-loading is performed on an index: (i) when first building the index (i.e. executing the CREATE INDEX statement), and (ii) when executing an explicit LOAD command. We will only detail the former in this section, but the principles to realize multi-valued bulk-loading are the same for both (for details on the latter, see [24]). Two data flows for the creation of a secondary index are illustrated in Fig. 5. The goal of a bulk-loading data flow in this context is to feed a sorted sequence of records to the LOAD operator, which will create the initial B+ tree. To establish a baseline, the left flow describes the data flow to bulk-load a traditional single-valued index on the field name inside the Stores dataset. We start at the bottom node SCAN, which will scan the primary index on Stores to extract the leading key value of our index, name, and the primary key of the Stores dataset, store_id. Next, we perform a sort using the key fields we just extracted. Finally, we feed the sorted <name, store_id> tuples to the LOAD operator.

On the right side of Fig. 5 we show the data flow to bulk-load a multi-valued index on the string values inside a categories array of the Stores dataset. There are two differences: (1) the inclusion of the UNNEST operator to extract the values inside the multi-valued field (these values are bound to the variable f in the figure), and (2) the inclusion of the DISTINCT operator to remove any duplicate B+ tree keys. More generally, any data flow to bulk-load a multi-valued index must extract two sets of values: the sort key values via a sequence of UNNEST operators, and the primary key values of the dataset associated with the index. Duplicate B+ tree key values are then removed (performed after the ORDER so as to execute a single-pass duplicate elimination) before being handed off to the LOAD operator. No changes are required to the LOAD operator itself.

5.3. Maintaining an index

Three types of dataset maintenance operations are offered by AsterixDB: (a) INSERT, (b) DELETE, and (c) UPSERT (to insert if the document does not exist, and to update it otherwise).

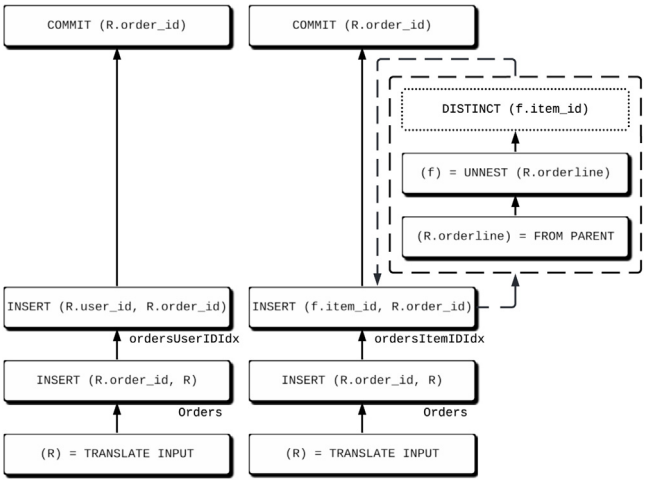


Fig. 6. Two separate data flows for performing an INSERT statement on a dataset. The left describes the data flow for executing an INSERT on a dataset with one single-valued index, while the right describes the data flow for executing an INSERT on a dataset with one multi-valued index.

5.3.1. Realizing a INSERT statement

The INSERT statement is one of three dataset maintenance operations offered by AsterixDB (the other two being DELETE and UPSERT). Before diving into the INSERT data flow, we must first address concurrency. The previous operation, bulk-loading, is always performed as a single *isolated* transaction. In contrast, queries and the aforementioned maintenance operations have no such security. To set the scene, transactions in AsterixDB are (i) of record-level granularity, (ii) local to each cluster node, and (iii) act across a dataset's primary and secondary indexes. Record-level locks are acquired to handle write operations (e.g. maintenance operations) on the primary index and they are held until the transaction itself commits [25]. If the lock to some primary index entry is granted to a transaction, no other operations from other transactions can be performed on that primary index entry until the former transaction commits. In contrast to primary indexes, locks are *not* acquired when accessing secondary indexes. No locking here means that a read operation on a secondary index can potentially read uncommitted data. To prevent inconsistencies between a data structure that requires locks (a primary index) and a data structure that does not (a secondary index), the index entries retrieved from the secondary index are first validated by fetching their corresponding records from the primary index before the entry itself is used by the rest of the transaction.

We will now describe the data flows to realize an INSERT statement, keeping these concurrency constraints in mind. The goal of a maintenance operation on a dataset is two-fold: to update the primary index of the dataset and to update all secondary indexes associated with the dataset. Two data flows for performing an INSERT statement are illustrated in Fig. 6.

The left data flow in Fig. 6 performs an INSERT statement on the Orders dataset with one traditional single-valued secondary index on user_id. We again start at the bottom node, where we will translate the documents given by the user into data records (assigned the variable 'R' in our figure). Next, we extract the primary key order_id associated with the dataset and insert the tuple <R.order_id, R> into the primary index of Orders. Primary index maintenance operations are always performed before any secondary index maintenance operations to prevent inconsistencies that could stem from other transactions on the secondary indexes themselves. After performing the primary index insertion, we extract the leading key field user_id from the record and insert the secondary index entry. Once we are done with this insert, we hand the primary key value to the COMMIT operator, that then releases the lock associated with that specific record. In contrast to the data flows for bulk-loading, notice that there are no *blocking* operators here. Once the primary index INSERT operator is finished with a single record (more accurately, a frame of records), it can hand the processed tuple(s) off to the following operator which will carry out its computation and perform the same hand off to its child operator. This compute + hand off process is repeated until the tuples all reach the COMMIT. This pipelined style of execution adheres to AsterixDB's record-level transaction semantics while releasing locks as early as possible without loss of isolation.

On the right of Fig. 6, we detail a similar scenario: we are performing an INSERT statement on the Orders dataset with one multi-valued secondary index on the item_id field inside objects of the orderline array of the Orders dataset. Similar to the single-valued case, we perform the insertion on the primary index before performing any secondary index insertions and conclude our data flow with the same COMMIT operator. The difference between the two data flows lies is the inclusion of a *subplan* attached to the secondary index INSERT operator itself. A subplan is an isolated DAG of operators that is used by the containing operator to perform some computation on a *single* value and then utilize the DAG's output for the containing operator's computation. For this particular subplan and containing operator INSERT, the value given to the DAG is an orderline array, the DAG's computation is extracting *distinct* item_id values from that particular array instance, and the containing operator's computation is pairing each item_id output with the record's primary key order_id and inserting this pair into the secondary index. Instead of eliminating duplicates via an explicit DISTINCT (as was the case for multi-valued index bulk loading), an *implicit* DISTINCT operation (denoted by the lack of shadow and dotted lines around the DISTINCT operator node in the figure) is performed by the storage layer via duplicate key rejection.

To motivate the use of subplans here, consider an alternative to the right data flow of Fig. 6, where subplans are not used and the UNNEST operator is inline with the rest of the plan. The first issue lies with the COMMIT operator, which uses the primary key of a record to conclude a transaction. UNNEST is a one-to-many operator, which will output one record for every item inside the array/multiset being UNNESTed. A scenario that might occur in this alternative data flow using the pipelined execution mentioned prior involves the primary key value reaching the COMMIT before the maintenance itself is finished. This scenario would cause a loss of transaction isolation, so in this alternative data flow we would need to add some form of blocking operator (e.g. ORDER) prior to the commit. Avoiding the inclusion of a blocking operator is the primary advantage of using the subplan data flow in Fig. 6. Compared to this alternative data flow, no blocking operators are introduced, reducing the time for which record locks are held and thus maximizing concurrency.

5.3.2. Realizing a DELETE statement

The next dataset maintenance operation we will discuss is the DELETE statement. There are two phases associated with DELETE: (i) a search phase (to find all qualifying records), and (ii) a delete phase. In AsterixDB, the storage-level API call to delete a tuple from an index is nearly identical to the API call to insert a tuple into an index. Consequently, the data flow for a DELETE statement is identical to the data flow for an INSERT statement, bar the inclusion of the search phase. To illustrate these similarities, suppose the DELETE statement in Listing 8 was issued to AsterixDB.

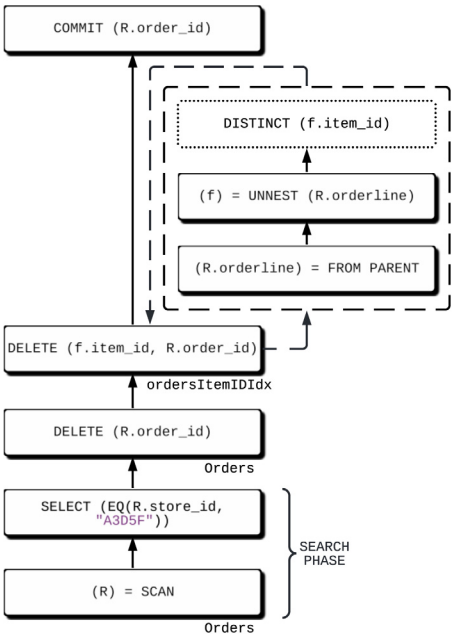


Fig. 7. Data flow for performing a DELETE statement on a dataset with a multi-valued index.

```
1 DELETE
2 FROM   Orders O
3 WHERE  O.store_id = "A3D5F";
```

Listing 8: DELETE statement associated with Fig. 7.

The data flow in Fig. 7 realizes Listing 8's deletion by first scanning the Orders dataset for records where the store_id is equal to "A3D5F". The primary key of a qualifying record (in our example, order_id) is then used to delete a record from the primary index. Next, the same subplan from Fig. 6 is used to extract distinct item_id values from the record's orderline array. Again, these item_id values are paired with the record's primary key and used to perform our maintenance operation. When all index entries associated with a qualifying record are deleted, the primary key is then handed off to the COMMIT operator to release the lock associated with that specific record.

5.3.3. Realizing an UPSERT statement

The third (and final) AsterixDB dataset maintenance operation is the UPSERT statement, where incoming records are either inserted or updated if they already exist. Updates to a record in AsterixDB are implemented by first deleting the old record, then inserting the new record. These deletions and insertions are performed at the storage-level, meaning that a data flow implementing UPSERT must now keep track of *two* values: the old record and the new record.

The data flow in Fig. 8 realizes an UPSERT statement by first translating the documents given by the user into data records (assigned the variable 'R_new' in our figure). Similar to the data flow for INSERT, we then extract the primary key of the dataset we are operating on to get the tuple <R_new.order_id, R_new>. We must now perform the UPSERT operation for our primary index using intra-operator-level actions (i.e. within the UPSERT operator block). To determine if we need to perform a deletion, we search our primary index using the primary key we just extracted. If we find an existing entry in our index, then we (1) perform a deletion for the old entry, (2) retrieve the old entry from our index (assigned the variable 'R_old' in our figure) to pass along to the following

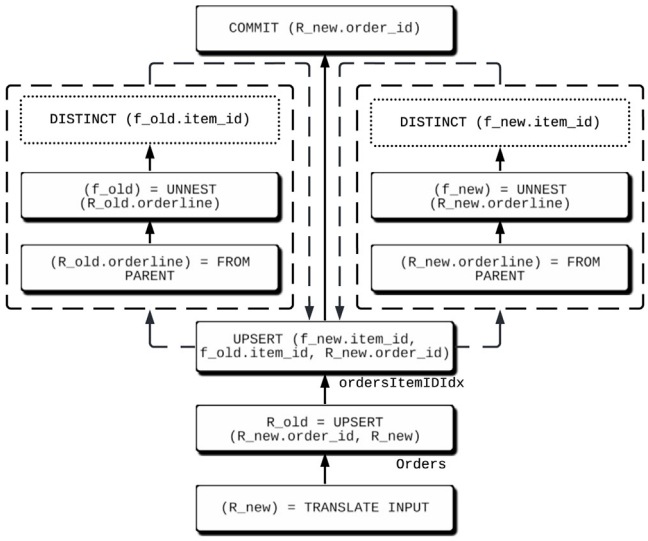


Fig. 8. Data flow for performing a UPSERT statement on a dataset with a multi-valued index.

operators, and (3) perform an insertion for the new entry. If we do not find an existing entry in our index, then we just perform an insertion for the new entry and assign a value of MISSING to R_old.

The next operator in Fig. 8 data flow is the UPSERT for our multi-valued index. If we find that R_old is MISSING, then the remainder of our data flow is identical to the data flow for INSERT: (1) extract item_id values from R_new.orderline, (2) pair distinct item_id values with the primary key R_new.order_id, (3) perform the insertion, and (4) pass R_new.order_id to the COMMIT operator to finish the transaction for that specific record. If we find that R_old is not MISSING, then we first extract distinct item_id values from R_old and distinct item_id values from R_new using two separate subplans. Similar to the data flow for INSERT, the use of subplans here allow us to exclude blocking operators from the UPSERT data flow. From here we have four possible actions, conditioned on the orderline field for both R_new and R_old:

1. If the field orderline exists in R_old but not R_new, then delete all associated f_old entries from the index.
2. If the field orderline exists in R_new but not in R_old, then insert all associated f_new entries into the index.
3. If the field orderline exists in both R_new and R_old but R_new $\neq$ R_old, then delete all associated f_old entries and insert all associated f_new entries.
4. If the field orderline exists in both R_new and R_old, and R_new = R_old, then no operations are performed.

When our multi-valued index UPSERT operator is finished with a record, the record's primary key is then given to the COMMIT operator to finish the transaction for that specific record.

5.4. Optimizing an indexable query

The goal of the AsterixDB query optimizer is to take an initial data flow (henceforth referred to as a query plan) and transform the query plan using a set of heuristics. The general heuristic discussed here involves replacing full dataset *scans* with a more selective *search* of the full dataset when applicable. This more selective search is enabled through the use of a secondary index.

We will begin by describing how multi-valued indexes can be utilized in query plans. Listing 9 describes an existential quantification query that aims to find all users that have an office phone.

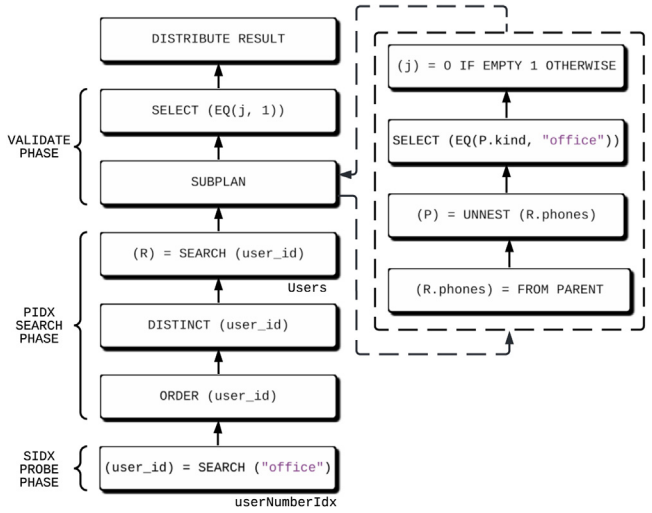


Fig. 9. Index leveraging plan to execute the existential quantification query in Listing 9.

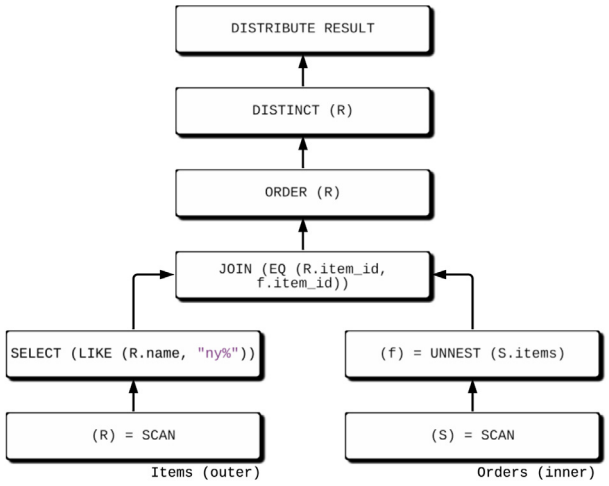


Fig. 10. Non-index-leveraging query plan to execute the array-probing join query in Listing 10.

```
1 FROM    User U
2 WHERE   SOME P IN U.phones
3         SATISFIES P.kind = "office"
4 SELECT  U;
```

Listing 9: A SQL++ existential quantification query.

If an applicable index exists (i.e., a multi-valued index on the `kind` field inside the `phones` array of objects), then the query plan in Fig. 9 would be generated. In this query plan, we divide the utilization of our index into three phases:

1. Secondary index probe phase (`SIDX_PROBE`)
2. Primary index search phase (`PIDX_PROBE`)
3. Validation phase (`VALIDATE`)

Starting from the bottom operator, we search our index for all entries that have a sort key equal to `"office"`. The output of this search is the primary key of the indexed dataset: `user_id`. We refer to this operator as the secondary index probe phase.

As a consequence of non-locking secondary indexes, records fetched by the probe phase may become invalid after the initial secondary index lookup. To ensure that only valid records are returned to the remainder of the plan after the index lookup, we require two additional phases: the primary index search phase, and the validation phase. In the primary index search phase, we remove duplicate primary key values via the `ORDER` and `DISTINCT` operators before using these values to search the primary index for the records associated with `Users`. The inclusion of the `ORDER` operator has the added benefit of minimizing the number of index lookups [26] (an `ORDER` operator exists in the same place for single-valued index leveraging query plans for this same reason). After fetching the qualifying records, we perform the validation phase using the two following operators: the `SUBPLAN` operator and the `SELECT` operator. The `SUBPLAN` operator contains a subplan to evaluate the indicator variable j, which is equal to 0 when a record's `phones` array does not contain an object whose `kind` field is equal to `"office"` and 1 otherwise. j is then attached to each record and handed off to the `SELECT` operator that will filter out all results where $j = 1$ (i.e. records that satisfy the existential quantification).

Utilization of a multi-valued index as described in Fig. 9 can also be extended to queries with a join that requires the values of a multi-valued field.

```
1 FROM    Items I
2 JOIN    (
3     FROM    Orders O
4     UNNEST  O.orderline OL
5     SELECT  OL.item_id AS item_id
6 ) OL
7 ON      OL.item_id /*+indexnl*/ = I.item_id
8 WHERE   I.name LIKE "ny%"
9 SELECT  DISTINCT I.i_id, I.name;
```

Listing 10: A join query that probes the inside of an multi-valued field of another dataset.

Listing 10 describes one such query, where we aim to find all items that start with `"ny"` and that are referenced in the `orderline` array of a `Orders` document. By default, AsterixDB will choose to evaluate joins using a hybrid hash approach, so we annotate our join predicate with '`/*+indexnl*/`' to inform the optimizer that we want to evaluate this join using an index if possible. If an applicable index does not exist (i.e. a multi-valued index on the item ID of an orderline), the plan illustrated in Fig. 10 is generated. Conceptually, we (a) `SCAN` the `Items` dataset to search for records that satisfy the `LIKE` predicate, (b) `SCAN` the `Orders` dataset to extract all `item_id` values from each document's `orderline` array, (c) perform an equi-join, and (d) deliver unique, qualifying `Items` records back to the user. If there are only a few qualifying records in our outer dataset `Items` and a massive amount of `orderline` documents in our inner dataset `Orders`, then the cost of this query plan is dominated by the `SCAN` of our inner dataset.

Now suppose that we do have a qualifying index for the query in Listing 10. The AsterixDB optimizer would recognize this and generate the plan illustrated in Fig. 11, which logically performs an index-nested loop join (INLJ). We divide the join using the same three phases from Fig. 9: a secondary index probe phase, a primary index probe phase, and a validation phase. Starting from the bottom two operators, we perform a search for qualifying records of the outer dataset `Items`. We then use the `item_id` field from this outer dataset to perform an index search and retrieve the primary key associated with our inner dataset `Orders`. These three operators compose the secondary index probe phase. The primary index search phase is comprised of the following three operators: an `ORDER` operator and a `DISTINCT` operator to remove duplicate primary key values, and then a `SEARCH` operator to retrieve qualifying records. Finally, the validation phase is performed by extracting the items from our multi-valued field and evaluating the join predicate to filter out invalid records.

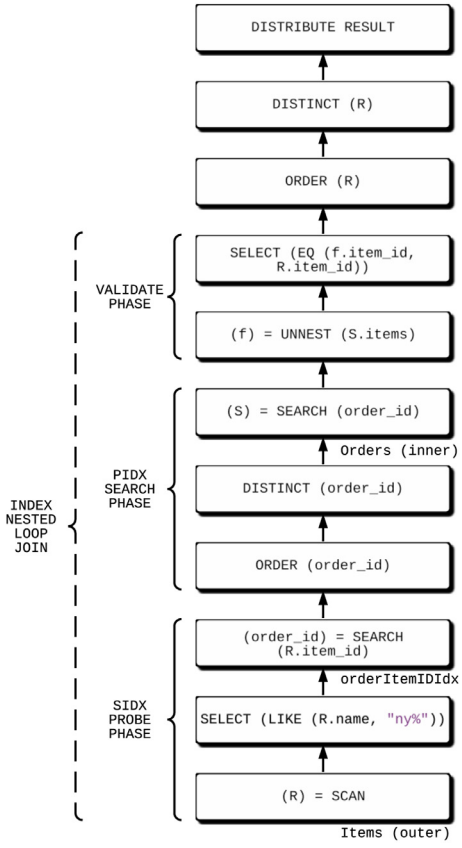


Fig. 11. Index-leveraging query plan to execute the array-probing join query in Listing 10.

Algorithm 1: Process for modifying an existing query plan to leverage an applicable multi-valued index.

Input: existing query plan Q, existing database indexes I
Output: **true** if Q has changed, **false** otherwise

for op := Preorder(*root of Q*) **do**
  if op *is neither a* SELECT *nor a* JOIN **then**
    continue;
  end
  op_S := first SCAN operator of Postorder(op);
  i := FindQualifyingMultiValuedIndex(I, op, op_S);
  if i *is* **null then**
    continue;
  end
  C := conjuncts from op that map to fields in i;
  PK := primary key variables from op_S;
  if op *is a* SELECT **then**
    SIDX_PROBE := (SEARCH$_{SIDX}$(C));
    PIDX_SEARCH := (ORDER(PK) → DISTINCT(PK) → SEARCH$_{PIDX}$(PK));
    VALIDATE := (op);
    replace (op_S) in Q with (SIDX_PROBE → PIDX_SEARCH → VALIDATE);
  end
  else
    SIDX_PROBE := (outer branch of op → SEARCH$_{SIDX}$(C));
    PIDX_PROBE := (ORDER(PK) → DISTINCT(PK) → SEARCH$_{PIDX}$(PK));
    VALIDATE := (inner branch of op without op_S → SELECT(predicate of op));
    replace (op) in Q with (SIDX_PROBE → PIDX_SEARCH → VALIDATE);
  end
  return true;
end
return false;

Having described why the plans in Figs. 9 and 11 are transactionally correct, we now describe our heuristic of replacing dataset scans with index searches, implemented as a rule in AsterixDB's query optimizer. This rule is given in Algorithm 1, which is repeatedly executed until the query plan itself does not change (i.e., until the rule returns false). Following Algorithm 1 and using Fig. 10 as the input Q, we start from the root DISTRIBUTE operator, and work our way down to the JOIN operator (now bound to the variable op). op_S here is the Orders SCAN operator, and i is a qualifying multi-valued index (in this case, i = orderItemIDIdx). An index qualifies to be used in a query plan if there are variables in op produced from op_S or its children that match the structure defined in i. Given the orderItemIDIdx index specification in Listing 5, we match the orderline field inside the UNNEST operator and the consequent access to the item_id field in the JOIN operator.

At this point in the rule, we can commit to modifying Q. We extract the relevant conjuncts C from op that can be accelerated with an index (R.item_id = f.item_id) and determine the primary key variables PK of our inner branch (order_id). Though we are only describing the JOIN case, the SELECT case can be reasoned about in a similar fashion. For both cases, we aim to find three subgraphs that correspond to the three phases from before. Returning to Fig. 10 example, the SIDX_PROBE subgraph is composed of the outer branch (Items SCAN and the original SELECT operator for R.name), and a SEARCH$_{SIDX}$ with the relevant conjuncts C. The PIDX_PROBE subgraph is composed of the ORDER, DISTINCT, and the SEARCH$_{PIDX}$ operators with the primary key variables PK as the input to all. The VALIDATE subgraph is composed of the inner branch without the op_S (just the UNNEST operator in this example) and a new SELECT with the original join predicate. Finally, we replace the original join operator op in our query plan with the appropriate composition of all three subgraphs.

Listing 10 was an example of existential quantification, though multi-valued indexes can also accelerate queries that involve universal quantification. Currently we impose an additional constraint requiring that a non-emptiness clause on the array/multiset being universally quantified on must exist, as empty arrays and multisets satisfy such predicates vacuously but will not be indexed; this constraint could be relaxed in the future by storing empty multi-valued fields in the index in some way and handling the empty case separately. The approach we take to leverage a multi-valued index for use in evaluating a universal quantification predicate is *exactly the same* as the approach taken to leverage a multi-valued index for use in evaluating an existential quantification predicate: replace the dataset scan with a multi-valued index search. We can easily prove that such a query plan transformation is valid, starting with a universal quantification on a multi-valued field F where $|F| > 0$:

$$U = \{\ \forall f \in F \mid P(f)\ \} \tag{1}$$

Given the predicate P in the universal quantification of Eq. (1), a multi-valued index B+ tree search to evaluate P (e.g., the secondary index probe phase of an index-nested loop join query plan) returns the primary keys of all records that would satisfy the existential quantification:

$$E = \{\ \exists f \in F \mid P(f)\ \} \tag{2}$$

All entries in U also exist in E, making U a subset of E itself. The problem at this point is identical to the issue of removing invalid

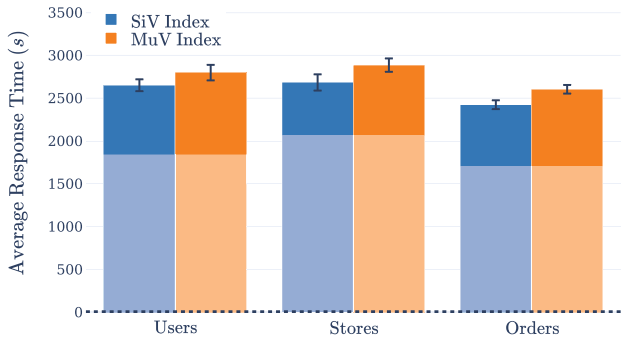


Fig. 12. Average response time for performing a `LOAD` to populate several indexed datasets. We compare executions that operate on a multi-valued indexed dataset (denoted as "MuV") and a single-valued indexed dataset (denoted as "SiV").

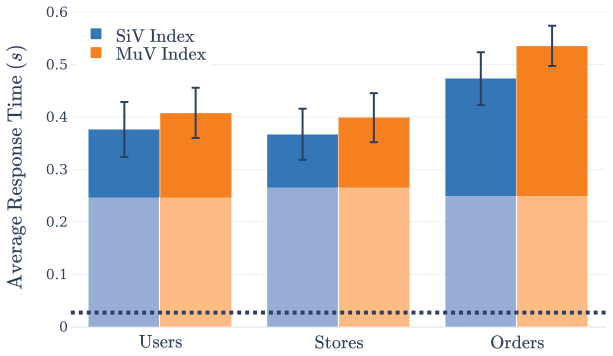


Fig. 13. Average response time for performing a 10,000 record `INSERT` on several indexed datasets.

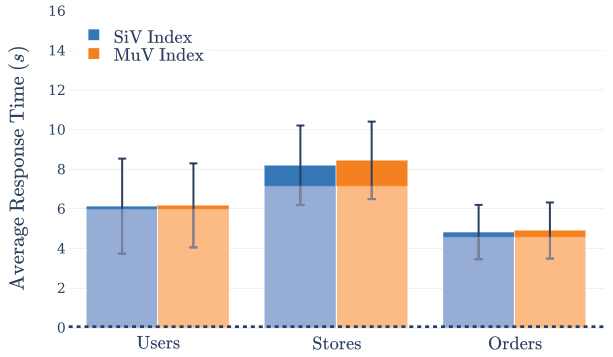


Fig. 14. Average response time for performing a 10,000 record `DELETE` on several indexed datasets.

records from the secondary index probe phase as a consequence of other concurrent transactions accessing the same secondary index. Hence, the following phases of primary index search and validation serve two purposes in the case of universal quantification: (i) remove invalid records that may have changed from the initial secondary index search, and (ii) remove entries in E that are not contained in U.

6. Evaluation

Our evaluation of multi-valued indexes is split into three parts: (1) evaluating the loading and maintenance cost (when compared to single-valued indexes), (2) evaluating multi-valued index-leveraging plans (when compared to full dataset scan plans), and (3) evaluating our implementation of multi-valued indexes against similar indexes in MongoDB and the Couchbase Query Service.

6.1. Index modification cost evaluation

In this section, we compare the cost of performing a modification operation (`LOAD`, `INSERT`, `DELETE`, or `UPSERT`) on a dataset with single-valued index(es) vs. the cost of performing a maintenance operation on a dataset with multi-valued index(es).

6.1.1. Experimental setup

All experiment runs were performed on a single-node AsterixDB instance, executed on an Intel Celeron J4125, 4 cores @ 2.7 GHz CPU with 8 GB of RAM and a single NGFF M.2 SSD. Each modification operation was executed on two separate instances of the datasets in Section 3.2: one where the dataset has only single-valued indexes, and another where the dataset has only multi-valued indexes. To normalize the cost associated with each storage layer write, the multi-valued fields in this experiment were restricted to contain a *single* item. All datasets used were larger than memory (`Users` being 15 GB @ 100 million records, `Stores` being 20 GB @ 90 million records, and `Orders` being 18 GB @ 65 million records), resulting in indexes that were larger than memory as well. For the `INSERT` and `UPSERT` experiment runs, 10,000 record chunks at a time were used until the dataset grew 0.5% in size. For the `DELETE` experiment runs, 10,000 record chunks were deleted at a time until the dataset shrunk 0.5% in size. To accelerate the search phase of the `DELETE` operation, a single-valued index on a 10,000 record chunk identifier was used. The response times shown will be for 10,000 record chunks. The data generator and statements used in this experiment can be found at https://github.com/glennga/ilima.

6.1.2. Results and analysis

Fig. 12 displays the average time to perform `LOAD` statements for the three datasets mentioned previously. The opaque (lower) portions of each bar graph represent the time to perform the `LOAD` statement without indexes (the insertion time is similar for both datasets). There does exist a slight slowdown for multi-valued indexes when compared to single-valued indexes, most emphasized in the dataset with two indexes (i.e. `Orders`). This slight difference in response time comes from the inclusion of the extra operators to extract the secondary key values and the overhead of using subplans.

Fig. 13 displays the average time to perform `INSERT` statements for all three datasets. The dotted black line represents a lower bound on the time to perform an `INSERT` statement, illustrating the time to compile the `INSERT` statement itself. The opaque (lower) portions of each bar graph represent the time to perform the `INSERT` statement on a non-indexed dataset (the insertion time is similar for both datasets). Again, there does exist a slight slowdown for multi-valued indexes when compared to single-valued indexes due to the secondary key value extraction.

Fig. 14 displays the average time to perform `DELETE` statements for each of our datasets. We note that the increase from statement execution time of milliseconds with `INSERT` statements to seconds with `DELETE` is due to the required search phase. The search phase for the `DELETE`s in this experiment involves (i) a secondary index search for the records with the appropriate chunk identifier, (ii) a primary index search for the 10,000 records from the secondary index search, and (iii) the secondary index validation step. The execution time difference between the single-valued indexed dataset and the multi-valued indexed dataset is not significant here, as the time to search for qualifying records outweighs the time to perform the actual deletion. The large opaque bars for each plot corroborate our observation. The cost of performing

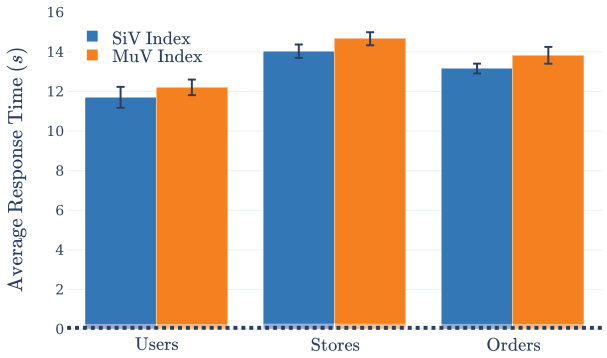


Fig. 15. Average response time for performing a 10,000 record UPSERT on several indexed datasets.

a record deletion for an (LSM) index in AsterixDB consists of a write to an in-memory data structure that will "reconcile" this deletion in the future, meaning that we do not directly touch the underlying index [23].

Fig. 15 describes the average time to perform UPSERT statements for our three datasets. All records being UPSERTed consist of new indexed fields, meaning that both an insertion and deletion will occur. Again, the cost of searching outweighs the time to perform the insertion and deletion. For both the INSERT and UPSERT experiments, records in the evaluated DML statements were *not* ordered by their primary key values. As noted in [26], sorting a collection of primary keys before searching the primary index potentially turns many small read operations into fewer and larger more efficient read operations. The unsorted nature of the records in the UPSERT statement explains the large deviation in DELETE times but not UPSERT times. The large difference in UPSERT times between the indexed and non-indexed instances (i.e. the solid bars vs. the barely visible opaque bars) is explained by an optimization AsterixDB takes: if there are no secondary indexes on a dataset, the output from a primary index UPSERT operator (`R_old`) is no longer used in the rest of the plan. Consequently, there is no need to fetch the old record from the primary index, massively accelerating the UPSERT operation.

6.2. Index vs. Full scan evaluation

In this section, we compare the performance of queries in AsterixDB that do not utilize an index vs. queries in AsterixDB that do utilize a multi-valued index.

6.2.1. Experimental setup

All experimental runs were performed on a single-node instance of AsterixDB, executed on an AWS c5.xlarge node, 4 vCPUs @ 3.4 GHz with 8 GB of RAM and AWS gp2 SSDs. The benchmark queries used here are from CH2 [27], a document-oriented combination of TPC-C and TPC-H designed for a HOAP (hybrid operational / analytical processing) workload. A total of 500 CH2 warehouses were generated for this experiment, resulting in three datasets larger than memory: `Orders` (25 GB), `Stock` (25 GB), and `Customer` (12 GB). Primary indexes were built on each dataset's primary key fields and one multi-valued index was built on the `delivery_d` field inside the `orderline` object array of the `Orders` dataset. For brevity, all fields in each query described here have their dataset prefix removed (e.g. `orderline` refers to `o_orderline` in the original set of queries). The full set of CH2 queries used in this experiment can be found in Appendix and at https://github.com/glennga/aconitum.

6.2.2. Results and analysis

Fig. 16 depicts the performance of several CH2 queries executed using query plans with and without a multi-valued index (i.e. a full dataset scan). We observe the median response times of the queries on the y-axis, and the selectivity of the index-applicable predicate (denoted as σ) on the logarithmic x-axis. σ represents the fraction of the dataset for which the predicate holds true. Beginning with the left of Fig. 16, we compare the performance of queries that do not execute any joins (i.e. only involve the `Orders` dataset). Query plans that use the multi-valued index achieve sub-second response times for $\sigma < 1.0e{-}5$, while query plans that perform a full scan of `Orders` consistently run longer than 4 min (a 350x speedup minimum). As σ grows larger and larger though, the response times for query plans that use the multi-valued index increases faster than their full scan counterparts. Beyond $\sigma > 1.7e{-}1$, we are better off evaluating queries 1, 6, and 12 using a full scan rather than using an index. As expected, plans that perform a secondary index search followed by a primary index search are vastly superior in response time to plans that perform full dataset scans when the applicable query predicate has a low selectivity [26].

The right graph in Fig. 16 tells a similar story with a multi-join query, query 7 (illustrated in Appendix A.3). Four plots are displayed here, varying the plan for CH query 7 in some way:

1. A query plan performing a full scan of `Orders` and a primary key index nested loop join (INLJ) to evaluate every join.
2. A query plan performing a full scan of `Orders` and a hash join (HJ) to evaluate every join.
3. A query plan using the multi-valued index on `Orders` and an INLJ to evaluate every join.
4. A query plan using the multi-valued index on `Orders` and a HJ to evaluate every join.

Starting with a comparison between plans (1) and (3) (using an INLJ and varying the use of the multi-valued index), we can reach the same conclusion as before: lower values of σ enable larger performance gains (i.e. speedup) when using a multi-valued index. However, when we compare plans (2) and (4) (using a HJ and varying the use of the multi-valued index), the overall speedup at low values of σ is significantly smaller (x2.75 for HJ vs. x110 for INLJ at $\sigma = 1.0e{-}5$). At low selectivity values for plan (4), the total response time is dominated by the time to perform every join (in particular, with the larger-than-memory datasets `Customer` and `Stock`). Even if the relevant `Orders` records can be retrieved in sub-second time, if we cannot accelerate the time to perform the joins then query 7 will always run longer than two minutes, regardless of whether we use a multi-valued index or not.

The main takeaway from this experiment is that multi-valued indexes can massively accelerate queries, but care should be taken to avoid using indexes to satisfy predicates on non-small σ values. As with single-valued indexes, AsterixDB (at the time of writing) does not vary its query plan based on different selectivity values. If an index can satisfy some predicate in a SELECT operator, AsterixDB will currently greedily default to integrate that index into the query plan regardless of σ unless a hint is provided to do otherwise. In terms of join methods, AsterixDB will default to hybrid-hash joins unless an index-nested loop join hint is provided.

6.3. System comparison evaluation

In this section, we compare the performance of queries that can benefit from multi-valued indexes running on AsterixDB, MongoDB, and the Couchbase Query Service (shortened to "Couchbase Query" for this section).

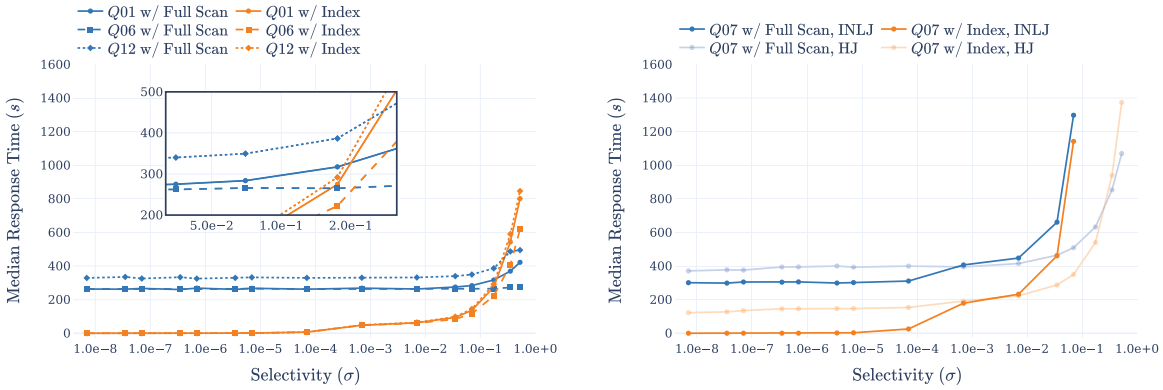


Fig. 16. Median response time vs. selectivity for several queries on an AsterixDB instance. We compare executions that utilize an multi-valued index, perform a full scan of the data, use INLJ to perform joins, and use hybrid-hash to perform joins.

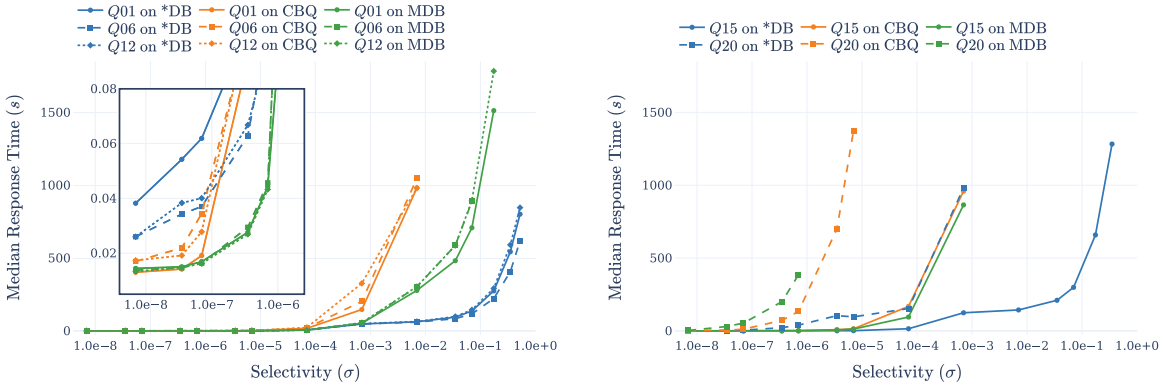


Fig. 17. Median response time vs. selectivity for two sets of queries across an AsterixDB instance (denoted as *DB), a MongoDB instance (denoted as MDB), and the query service on a Couchbase instance (denoted as CBQ).

6.3.1. Experimental setup

All experimental runs were performed on single-node instances of AsterixDB (version 0.9.7), MongoDB (version 4.4.6) and Couchbase Server (version Enterprise 7.0.0-beta), executed on an AWS c5.xlarge node, 4 vCPUs @ 3.4 GHz with 8 GB of RAM and AWS gp2 SSDs. The same CH2 benchmark was used for this experiment as well, using the same CH2 parameters (i.e. 500 total warehouses). For all systems, primary indexes were built on each dataset's primary key fields and one multi-valued index was built on the `delivery_d` field inside the `orderline` object array of the `Orders` dataset. To aid in the evaluation of query 20, a secondary index was also created for all systems on the `i_id` field of `Stock`. Each system was given 30 min maximum to execute each query as ad-hoc (i.e. not prepared) with the system terminating the query execution if it exceeded this maximum. All queries were written to leverage the multi-valued index and to use INLJ. The full set of CH2 queries used in this experiment can be found in Appendix and at https://github.com/glennga/aconitum.

6.3.2. Results and analysis

Fig. 17 depicts the selectivity vs. the response time of several queries executed using multi-valued indexes on different document databases. In the left graph, we have queries 1, 6, and 12 (the same queries used in the left graph of Fig. 16). At $\sigma \leq$ 1.0e–5, we observe the following response time hierarchy: MongoDB $\leq$ Couchbase Query $<$ AsterixDB. Query 1 on Couchbase Query actually has the smallest median response time with 13 ms at $\sigma =$ 7.0e–8, but queries 6 and 12 on Couchbase Query run roughly 3 ms slower than queries 6 and 12 on MongoDB at the same σ. At these low selectivity values, AsterixDB executes the slowest with a minimum response time of 25 ms. Each system achieving sub-second execution time for $\sigma <$ 3.5e–6 is not surprising, given the similarity in execution plans to leverage multi-valued indexes in each system. Take query 1, where each system starts by searching their respective multi-valued index for primary keys of `Orders` such that the indexed `delivery_d` field is between the two variables. Duplicate primary keys are then removed by consulting the primary data source for `Orders` documents. From here, the `orderline` array undergoes an `UNNEST` operation. AsterixDB differs from the remaining two here in that the secondary index entries must be validated, so a filter is performed on `delivery_d` using our two variables. Documents at this point for all plans are then grouped by the `number` field of their `orderline` document and the aggregates are computed. Finally, the groups are sorted by their `number` field.

Another observation we can draw from this experiment is how resilient each system's response time is (per query) to increasing values of σ. In particular, we are interested in finding the range of selectivities each system supports for each query (i.e. how many query executions that are below the 30 min timeout). Under this light, a longer and flatter plot is more ideal. Couchbase Query was able to execute queries 1, 6, and 12 up to $\sigma =$ 7.0e–3. MongoDB has larger range here of selectivities here: query 6 up to $\sigma =$ 7.0e–2 and queries 1 and 12 up to $\sigma =$ 1.75e–1. We can conclude here that AsterixDB is the most resilient to large selectivity values for queries 1, 6, and 12, capping out at $\sigma =$ 5.25e–1.

In the right graph of Fig. 17, we show the median response time vs. the selectivity for two multi-join queries, queries 15 and 20. To roughly describe the complexity of each query, we list the joins each query contains. Query 15 includes two joins:

Orders ⋈ Stock and Stock ⋈ Supplier. Query 20 includes four joins: Orders ⋈ Stock, Item ⋈ Stock, Stock ⋈ Supplier, and Supplier ⋈ Nation. In the face of these more complex queries (in contrast to the single dataset queries 1, 6, and 12), how resilient are each system's response times for increasing values of σ? With query 15, both MongoDB and Couchbase Query were able to execute for increasing values of σ until $\sigma = 7.0e{-}7$ before timing out, while AsterixDB was able to execute up to $\sigma = 3.5e{-}1$. With query 20, we observe the smallest range of σ values thus far across any system for MongoDB: a maximum of $\sigma = 7.0e{-}7$ before executing beyond 30 min. Couchbase Query comes in second: a maximum of $\sigma = 7.0e{-}6$ before exceeding the timeout. Finally, AsterixDB again demonstrates the most resiliency in response time, executing up to $\sigma = 7.0e{-}4$ before the executing beyond 30 min.

7. Conclusion

We have described the various steps needed to support secondary indexes for multi-valued fields in AsterixDB. In short, this consisted of: (i) detailing an approach that separated the implementation of multi-valued indexes with the low-level storage layer of a database, (ii) describing a multi-valued index specification language that is neither ambiguous with respect to structure nor verbose, (iii) explaining the foundations for bulk loading and maintaining multi-valued indexes, (iv) illustrating the evaluation for existential and universal quantification queries, and (v) describing three sets of experiments that evaluated the maintenance cost and efficacy of multi-valued indexes in AsterixDB. We also like to stress that although this work was done in AsterixDB, these concepts can easily be applied to other systems with data flow and storage layers. We would also like to mention that the Couchbase Analytics Service, which uses the Apache AsterixDB query engine internally [28], now includes this version of multi-valued indexes [29].

Potential future work with respect to AsterixDB multi-valued indexes involves (a) applying these same concepts to accelerate other types of indexes (e.g. R Trees), (b) storing NULL values in multi-valued indexes with composite keys to accelerate queries that only involve a prefix of the sort key, and (c) storing empty arrays and multisets in order to accelerate general universal quantification queries (i.e. remove the need for a non-emptiness clause).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to acknowledge several individuals from Couchbase for their help with this research. We want to thank Dmitry Lychagin for providing input on the index specification syntax as well as direction on other AsterixDB specifics. We also want to thank Vijay Sarathy for his help with the CH2 benchmark and tuning the Couchbase Query service. This research was supported in part by National Science Foundation, USA awards IIS-1838248, IIS-1954962, and CNS-1925610, by the HPI Research Center in Machine Learning and Data Science at UC Irvine, and by the Donald Bren Foundation (via a Bren Chair).

Appendix. CH2 queries

In this appendix, we show all of the CH2 queries that were used for the experiments in Section 6.3. To modify the selectivity for each experiment, the dates $D1 and $D2 in each query below were varied.

A.1. Query 1

Report the total amount and quantity of all shipped order-lines between dates $D1 and $D2. Additionally report the average amount and quantity, and the total count of all order-lines ordered by the individual order-line number.

AsterixDB (SQL++)

```
FROM     Orders O, O.orderline OL
WHERE    OL.delivery_d BETWEEN $D1 AND $D2
GROUP BY OL.number
SELECT   OL.number,
         SUM(OL.quantity) AS sum_qty,
         SUM(OL.amount) AS sum_amount,
         AVG(OL.quantity) AS avg_qty,
         AVG(OL.amount) AS avg_amount,
         COUNT(*) AS count_order
ORDER BY OL.number;
```

Couchbase Query Service (N1QL for Query)

```
FROM     Orders O
UNNEST   O.orderline OL
WHERE    OL.delivery_d BETWEEN $D1 AND $D2
GROUP BY OL.number
SELECT   OL.number,
         SUM(OL.quantity) AS sum_qty,
         SUM(OL.amount) AS sum_amount,
         AVG(OL.quantity) AS avg_qty,
         AVG(OL.amount) AS avg_amount,
         COUNT(*) AS count_order
ORDER BY OL.ol_number;
```

MongoDB (MQL Aggregate Pipeline)

```
{ $match: {orderline: {
  $elemMatch: {
    delivery_d: {$gte: $D1, $lte: $D2}}}},
{ $unwind: {path: $orderline} },
{ $group: {
    _id: $orderline.number,
    sum_qty: {$sum: $orderline.quantity},
    sum_amount: {$sum: $orderline.amount},
    avg_qty: {$avg: $orderline.quantity},
    avg_amount: {$avg: $orderline.amount},
    count_order: {$sum: 1}}},
{ $sort: {orderline.number: 1} }
```

A.2. Query 6

List the total amount of archived revenue from order-lines that were delivered between $D1 and $D2 and with quantity between 1 and 100,000.

AsterixDB (SQL++)

```
FROM   Orders O, O.orderline OL
WHERE  OL.delivery_d BETWEEN $D1 AND $D2 AND
       OL.quantity BETWEEN 1 AND 100000
SELECT SUM(OL.amount) AS revenue;
```

Couchbase Query Service (N1QL for Query)

```
FROM   Orders O
UNNEST O.orderline OL
WHERE  OL.delivery_d BETWEEN $D1 AND $D2 AND
       OL.quantity BETWEEN 1 AND 100000
SELECT SUM(OL.amount) AS revenue;
```

MongoDB (MQL Aggregation Pipeline)

```
{ $match: {orderline: {
  $elemMatch: {
   delivery_d: {$gte: $D1, $lte: $D2}}}},
{ $match: {orderline.quantity: {
   $gte: 1,
   $lte: 100000}}},
{ $group: {
   _id: None,
   revenue: {$sum: $orderline.amount}}}
```

A.3. Query 7

Show the bi-directional trade volume between Germany and Cambodia within dates $D1 and $D2, sorted by the nation name and the considered years.

AsterixDB (SQL++)

```
FROM      Orders O, O.orderline OL, Stock S,
          Customer C, Supplier SU, Nation N1,
          Nation N2
LET       suppkey = ((S.s_w_id * S.s_i_id) % 10000),
          nationkey = STRING_TO_CODEPOINT(
            SUBSTR(C.state, 1, 1))[0]
WHERE     S.w_id = OL.supply_w_id AND
          S.i_id = OL.i_id AND
          C.id = O.c_id AND
          C.w_id = O.w_id AND
          C.d_id = O.d_id AND
          SU.suppkey = suppkey AND
          N1.nationkey = SU.nationkey AND
          N2.nationkey = nationkey AND
          ( ( N1.name = 'Germany' AND
              N2.name = 'Cambodia' ) OR
            ( N1.name = 'Cambodia' AND
              N2.name = 'Germany' ) ) AND
          OL.delivery_d BETWEEN $D1 AND $D2
GROUP BY  SU.nationkey, nationkey,
          SUBSTR(O.entry_d, 0, 4)
SELECT    SU.nationkey AS supp_nation,
          nationkey AS cust_nation,
          SUBSTR(O.entry_d, 0, 4) AS l_year,
          SUM(OL.amount) AS revenue
ORDER BY  SU.nationkey, cust_nation, l_year;
```

Couchbase Query Service (N1QL for Query)[1]

```
FROM      Orders O
UNNEST    O.orderline OL
JOIN      Stock S
ON        OL.supply_w_id = S.w_id AND
          OL.i_id = S.i_id
JOIN      Customer C
ON        C.id = O.c_id AND
          C.w_id = O.w_id AND
          C.d_id = O.d_id
JOIN      Supplier SU
ON        ((S.w_id * S.i_id) % 10000) = SU.suppkey
JOIN      Nation N1
ON        SU.nationkey = N1.nationkey
JOIN      Nation N2
ON        (stringToCodepoint(
            SUBSTR(C.state, 1, 1)))[0]
            = N2.nationkey
LET       nationkey = (stringToCodepoint(
            SUBSTR(C.state, 1, 1)))[0]
WHERE     OL.delivery_d BETWEEN $D1 AND $D2 AND
          ( ( N1.name = 'Germany' AND
              N2.name = 'Cambodia' ) OR
            ( N1.name = 'Cambodia' AND
              N2.name = 'Germany' ) )
GROUP BY  SU.nationkey, nationkey,
          SUBSTR(O.entry_d, 0, 4)
SELECT    SU.nationkey AS supp_nation,
          nationkey AS cust_nation,
          SUBSTR(O.entry_d, 0, 4) AS l_year,
          SUM(O.amount) AS revenue
ORDER BY  SU.nationkey, cust_nation, l_year;
```

MongoDB (MQL Aggregation Pipeline)

```
{ $match: {orderline: {
  $elemMatch: {delivery_d: {
   $gte: $D1,
   $lte: $D2}}}},
{ $unwind: {path: $orderline] },
{ $lookup: {
   from: Stock,
   localField: orderline.i_id,
   foreignField: i_id,
   as: stock} },
{ $unwind: {path: $stock} },
{ $match: {$expr: {
   $eq: [$orderline.supply_w_id,
         $stock.w_id]}}},
{ $lookup: {
   from: Customer,
   localField: id,
   foreignField: id,
   as: customer} },
{ $unwind: {path: $customer} },
{ $match: {$expr: {
   $and: [{$eq: [$customer.w_id,
                 $w_id]},
          {$eq: [$customer.d_id,
                 $d_id]}]}}},
{ $addFields: {
    supplier_no: {$mod: [
      {$multiply: [$stock.w_id,
                   $stock.i_id]}, 10000]}}},
{ $lookup: {
   from: Supplier,
   localField: supplier_no,
   foreignField: suppkey,
   as: supplier} },
{ $unwind: {path: $supplier} },
{ $lookup: {
   from: Nation,
   localField: supplier.nationkey,
   foreignField: nationkey,
   as: nation1} },
{ $unwind: {path: $nation1} },
{ $addFields: {
   nationkey: {$function: {
    body: 'function(inputString)
           { return inputString.codePointAt(0); }',
    args: [{$substr: [$customer.state, 1, 1]}],
    lang: js}}}},
{ $lookup: {
   from: Nation,
   localField: nationkey,
   foreignField: nationkey,
   as: nation2} },
{ $unwind: {path: $nation2} },
{ $match: {$expr: {
   $or: [
    {$and: [
     {$eq: [$nation1.name, 'Germany']},
     {$eq: [$nation2.name, 'Cambodia']}]},
    {$and: [
     {$eq: [$nation1.name, 'Cambodia']},
     {$eq: [$nation2.name, 'Germany']}]}]}}},
{ $group: {
   _id: {
    supp_nation: $supplier.nationkey,
    cust_nation: $nationkey,
    l_year: {$substr: [entry_d, 0, 4]}},
   revenue: {$sum: $orderline.amount}}},
{ $sort: {supp_nation: 1, cust_nation: 1, l_year: 1} }
```

A.4. Query 12

Count the number of high and low priority orders between dates $D1 and $D2, grouped by the number of order-lines in each order.

[1] The function `stringToCodepoint` represents an external user-defined function that returns the code-point for the first character of a string.

AsterixDB (SQL++)

```
FROM      Orders O, O.orderline OL
WHERE     O.entry_d <= OL.delivery_d AND
          OL.delivery_d BETWEEN $D1 AND $D2
GROUP BY O.ol_cnt
SELECT    O.ol_cnt,
          SUM(CASE WHEN O.carrier_id = 1 OR
                        O.carrier_id = 2
              THEN 1 ELSE 0 END) AS high_line_count,
          SUM(CASE WHEN O.carrier_id <> 1 OR
                        O.carrier_id <> 2
              THEN 1 ELSE 0 END) AS low_line_count
ORDER BY O.ol_cnt;
```

Couchbase Query Service (N1QL for Query)

```
FROM      Orders O
UNNEST    O.orderline OL
WHERE     O.entry_d <= OL.delivery_d AND
          OL.delivery_d BETWEEN $D1 AND $D2
GROUP BY O.ol_cnt
SELECT    O.ol_cnt,
          SUM(CASE WHEN O.carrier_id = 1 OR
                        O.carrier_id = 2
                   THEN 1 ELSE 0 END)
                   AS high_line_count,
          SUM(CASE WHEN O.carrier_id <> 1 OR
                        O.carrier_id <> 2
                   THEN 1 ELSE 0 END)
                   AS low_line_count
ORDER BY O.ol_cnt;
```

MongoDB (MQL Aggregation Pipeline)

```
{ $match: {orderline: {
    $elemMatch: {
      delivery_d: {$gte: $D1, $lte: $D2}}}} },
{ $match: {$expr: {
    $lte: [$entry_d,
           $orderline.delivery_d]}} },
{ $project: {
    ol_cnt: $ol_cnt,
    high_line: {
     $switch: {
      branches: [
       {case: {$in: [$carrier_id, [1, 2]]},
        then: 1}],
     default: 0}},
    low_line: {
     $switch: {
      branches: [
       {case: {$in: [$carrier_id, [1, 2]]},
        then: 0}],
      default: 1}}} },
{ $group: {
    _id: $ol_cnt,
    high_line_count: {$sum: high_line},
    low_line_count: {$sum: low_line}} },
{ $sort: {ol_cnt: 1} }
```

A.5. Query 15

Find the top supplier or suppliers who contributed the most to the overall revenue for items shipped between dates $D1 and $D2.

AsterixDB (SQL++)

```
WITH      Revenue AS (
 FROM      Stock S, Orders O, O.orderline OL
 WHERE     OL.i_id = S.i_id AND
           OL.supply_w_id = S.w_id AND
           OL.delivery_d BETWEEN $D1 AND $D2
 GROUP BY ((S.w_id * S.i_id) % 10000)
 SELECT    ((S.w_id * S.i_id) % 10000) AS supplier_no,
           SUM(OL.amount) AS total_revenue
)
FROM      Supplier SU, Revenue R
WHERE     SU.suppkey = R.supplier_no AND
          R.total_revenue = (
           FROM    Revenue
           SELECT VALUE MAX(total_revenue)
          )[0]
SELECT    SU.suppkey,
          SU.name,
          SU.address,
          SU.phone,
          R.total_revenue
ORDER BY SU.suppkey;
```

Couchbase Query Service (N1QL for Query)

```
WITH      Revenue AS (
          FROM      Orders O
          UNNEST    O.orderline OL
          JOIN      Stock S
          ON        OL.i_id = S.i_id AND
                    OL.supply_w_id = S.w_id
          WHERE     OL.delivery_d BETWEEN $D1 AND $D2
          GROUP BY ((S.w_id * S.i_id) % 10000)
          SELECT    ((S.w_id * S.i_id) % 10000)
                    AS supplier_no,
                    SUM(OL.amount) AS total_revenue
)
FROM      Revenue R
JOIN      Supplier SU
ON        SU.suppkey = R.supplier_no
WHERE     R.total_revenue = (
           FROM    Revenue M
           SELECT VALUE MAX (M.total_revenue)
)[0]
SELECT    SU.suppkey, SU.name, SU.address, SU.phone,
          R.total_revenue
ORDER BY SU.suppkey;
```

MongoDB (MQL Aggregation Pipeline)

```
{ $match: {orderline: {
    $elemMatch: {delivery_d: {
     $gte: $D1,
     $lte: $D2}}}} },
{ $unwind: {path: $orderline} },
{ $lookup: {
    from: Stock,
    localField: orderline.i_id,
    foreignField: i_id,
    as: stock} },
{ $unwind: {path: $stock} },
{ $match: {$expr: {
    $eq: [$orderline.supply_w_id,
          $stock.w_id]}} },
{ $project: {
    supplier_no: {$mod: [{$multiply:
     [$stock.w_id, $stock.i_id]}, 10000]},
    amount: $orderline.amount} },
{ $group: {
    _id: $supplier_no,
    total_revenue: {$sum: $amount}} },
{ $lookup: {
    from: Supplier,
    localField: _id,
    foreignField: suppkey,
    as: supplier} },
{ $unwind: {path: $supplier} },
{ $group: {
    _id: None,
    data: {$push: $$ROOT},
    max_revenue: {$max: $total_revenue}} },
{ $unwind: $data },
{ $match: {$expr: {
    $eq: [$data.total_revenue,
          $max_revenue]}} },
{ $project: {
    suppkey: $data.supplier.suppkey,
    name: $data.supplier.name,
    address: $data.supplier.address,
    phone: $data.supplier.phone,
    total_revenue: $data.total_revenue} },
{ $sort: {suppkey: 1} }
```

A.6. Query 20

Find suppliers in Germany having selected parts (whose data has a prefix `'co'`) that may be candidates for a promotional offer if the quantity of these items is more than 50% of the total quantity which has been ordered between `$D1` and `$D2`.

AsterixDB (SQL++)

```
FROM     Supplier SU, Nation N
WHERE    SU.suppkey IN (
          FROM     Stock S, Orders O, O.orderline OL
          WHERE    S.i_id IN (
                    FROM   Item I
                    WHERE  I.data LIKE 'co%'
                    SELECT VALUE I.id
                   ) AND
                   OL.i_id = S.i_id AND
                   OL.delivery_d BETWEEN $D1 AND $D2
          GROUP BY S.i_id, S.w_id, S.quantity
          HAVING   (100 * S.quantity) >
                   SUM(OL.quantity)
          SELECT   VALUE ((S.w_id * S.i_id) % 10000)
         ) AND
         SU.nationkey = N.nationkey AND
         N.name = 'Germany'
SELECT   SU.name,
         SU.address
ORDER BY SU.name;
```

Couchbase Query Service (N1QL for Query)

```
WITH     SupplierKeys AS (
          FROM     Orders O
          UNNEST   O.orderline OL
          JOIN     Stock S
          USE      HASH(BUILD)
          ON       OL.i_id = S.i_id
          JOIN     Item I
          ON       I.id = S.i_id
          WHERE    I.data LIKE 'co%' AND
                   OL.delivery_d BETWEEN $D1 AND $D2
          GROUP BY S.i_id, S.w_id, S.quantity
          HAVING   (100 * S.quantity) >
                   SUM(OL.quantity)
          SELECT   VALUE ((S.w_id * S.i_id) % 10000)
         )
FROM     SupplierKeys SK
JOIN     Supplier SU
ON       SU.suppkey = SK
JOIN     Nation N
ON       N.nationkey = SU.nationkey
WHERE    N.name = 'Germany'
SELECT   SU.name, SU.address
ORDER BY SU.name;
```

MongoDB (MQL Aggregation Pipeline)

```
{ $match: {orderline: {
    $elemMatch: {delivery_d: {
      $gte: $D1,
      $lte: $D2}}}} },
{ $unwind: {path: $orderline} },
{ $lookup: {
    from: Stock,
    localField: orderline.i_id,
    foreignField: i_id,
    as: stock} },
{ $unwind: {path: $stock} },
{ $lookup: {
    from: Item,
    localField: stock.i_id,
    foreignField: id,
    as: item} },
{ $unwind: {path: $item} },
{ $match: {item.data: {$regex: ^co}} },
{ $group: {
    _id: {
      i_id: $stock.i_id,
      w_id: $stock.w_id,
      quantity: $stock.quantity},
    total_quantity: {
      $sum: $orderline.quantity}} },
{ $match: {$expr: {
    $gt: [{$multiply: [100, $_id.quantity]},
          $total_quantity]}} },
{ $project: {
    supplier_no: {$mod:
      [{$multiply: [$_id.w_id, $_id.i_id]},
       10000]}} },
{ $lookup: {
    from: Supplier,
    localField: supplier_no,
    foreignField: suppkey,
    as: supplier} },
{ $unwind: {path: $supplier} },
{ $lookup: {
    from: Nation,
    localField: nationkey,
    foreignField: nationkey,
    as: nation} },
{ $unwind: {path: $nation} },
{ $match: {name: 'Germany'} },
{ $project: {
    name: $supplier.name,
    address: $supplier.address} },
{ $sort: {name: 1} }
```

References

[1] E. Bertino, W. Kim, Indexing techniques for queries on nested objects, IEEE Trans. Knowl. Data Eng. 1 (2) (1989) 196–214, http://dx.doi.org/10.1109/69.87960.

[2] P. Valduriez, Join indices, ACM Trans. Database Syst. 12 (2) (1987) 218–246, http://dx.doi.org/10.1145/22952.22955.

[3] E. Bertino, P. Foscoli, Index organizations for object-oriented database systems, IEEE Trans. Knowl. Data Eng. 7 (2) (1995) 193–209, http://dx.doi.org/10.1109/69.382292.

[4] A. Kemper, G. Moerkotte, Access support in object bases, SIGMOD Rec. 19 (2) (1990) 364–374, http://dx.doi.org/10.1145/93605.98745.

[5] O. Deux, The story of O2, IEEE Trans. Knowl. Data Eng. 2 (1) (1990) 91–108, http://dx.doi.org/10.1109/69.50908.

[6] D. Maier, J. Stein, Indexing in an object-oriented DBMS, in: Proceedings on the 1986 International Workshop on Object-Oriented Database Systems, OODS '86, IEEE Computer Society Press, Washington, DC, USA, 1986, pp. 171–182.

[7] D. Maier, J. Stein, A. Otis, A. Purdy, Development of an object-oriented DBMS, in: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '86, Association for Computing Machinery, New York, NY, USA, 1986, pp. 472–482, http://dx.doi.org/10.1145/28697.28746.

[8] W. Kim, J.F. Garza, N. Ballou, D. Woelk, Architecture of the ORION next-generation database system, IEEE Trans. Knowl. Data Eng. 2 (1) (1990) 109–124, http://dx.doi.org/10.1109/69.50909.

[9] K. Goczyla, The Partial-Order Tree: a new structure for indexing on complex attributes in object-oriented databases, in: EUROMICRO 97. Proceedings of the 23rd EUROMICRO Conference: New Frontiers of Information Technology (Cat. No.97TB100167), IEEE, Budapest, Hungary, 1997, pp. 47–54, http://dx.doi.org/10.1109/EURMIC.1997.617215.

[10] M. Nicola, B.V. der Linden, Native XML support in DB2 universal database, in: K. Böhm, C.S. Jensen, L.M. Haas, M.L. Kersten, P. Larson, B.C. Ooi (Eds.), Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005, ACM, Trondheim, Norway, 2005, pp. 1164–1174.

[11] Couchbase, Array indexing, 2021, Available at https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/indexing-arrays.html.

[12] MongoDB, Multikey indexes, 2021, Available at https://docs.mongodb.com/manual/core/index-multikey/.

[13] Oracle, Indexing arrays, 2021, Available at https://docs.oracle.com/en/database/other-databases/nosql-database/12.2.4.5/java-driver-table/indexing-arrays.html.

[14] S. Dew, Flex indexes, 2021, Available at https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/flex-indexes.html.

[15] S. Brucherseifer, Index basics, 2021, Available at https://www.arangodb.com/docs/stable/indexing-index-basics.html.

[16] R. Loveland, Inverted indexes, 2021, Available at https://www.cockroachlabs.com/docs/v20.2/inverted-indexes.

[17] O. Corporation, 13.1.15 CREATE INDEX statement, 2021, Available at https://dev.mysql.com/doc/refman/8.0/en/create-index.html.

[18] T. Sigaev, O. Bartunov, Gin for PostgreSQL, 2008, Available at http://www.sai.msu.su/~megera/wiki/Gin.
[19] AsterixDB, Apache AsterixDB, a scalable open source big data management system (BDMS), 2021, Available at https://asterixdb.apache.org.
[20] M.J. Carey, AsterixDB mid-flight: A case study in building systems in academia, in: 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019, pp. 1–12.
[21] V. Borkar, Y. Bu, E.P. Carman, N. Onose, T. Westmann, P. Pirzadeh, M.J. Carey, V.J. Tsotras, Algebricks: A data model-agnostic compiler backend for big data languages, in: Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 422–433, http://dx.doi.org/10.1145/2806777.2806941.
[22] V. Borkar, M. Carey, R. Grover, N. Onose, R. Vernica, Hyracks: A flexible and extensible foundation for data-intensive computing, in: Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, ICDE '11, IEEE Computer Society, USA, 2011, pp. 1151–1162, http://dx.doi.org/10.1109/ICDE.2011.5767921.
[23] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M.J. Carey, M. Dreseler, C. Li, Storage management in AsterixDB, Proc. VLDB Endow. 7 (10) (2014) 841–852, http://dx.doi.org/10.14778/2732951.2732958.
[24] G. Galvizo, On Indexing Multi-Valued Fields in AsterixDB, Computer Science Department, University of California, Irvine, 2021.
[25] Y.-S. Kim, Transactional and Spatial Query Processing in the Big Data Era (Ph.D. thesis), University of California, Irvine, 2016, pp. 22–56.
[26] G. Graefe, Modern B-Tree techniques, Found. Trends Databases 3 (4) (2011) 203–402, http://dx.doi.org/10.1561/1900000028.
[27] M. Carey, D. Lychagin, M. Muralikrishna, V. Sarathy, T. Westmann, CH2: A hybrid operational/analytical processing benchmark for NoSQL, in: 13th TPC Technology Conf. on Performance Evaluation & Benchmarking (TPC TC), TPCTC, 2021, pp. 62–80.
[28] M.A. Hubail, A. Alsuliman, M. Blow, M.J. Carey, D. Lychagin, I. Maxon, T. Westmann, Couchbase Analytics: Noetl for scalable NoSQL data analysis, Proc. VLDB Endow. 12 (2019) 2275–2286.
[29] Couchbase, Release notes for Couchbase Server 7.1, 2022, Available at https://docs.couchbase.com/server/current/release-notes/relnotes.html.