# On Multi-Valued Indexing in AsterixDB

Glenn Galvizo
ggalvizo@uci.edu
Department of Computer Science
Irvine, California, USA

Michael J. Carey
mjcarey@uci.edu
Department of Computer Science
Irvine, California, USA

## 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 compare AsterixDB multi-valued indexes against similar indexes in MongoDB and Couchbase Query.

## KEYWORDS

Multi-Valued Indexing, Index Specification, Index Implementation, Query Optimization, AsterixDB

CEUR-WS.org/Vol-3130/paper2.pdf

## 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 *AsterixDB*. 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.
(2) A multi-valued index specification language that is neither ambiguous with respect to structure nor verbose.
(3) Foundations for bulk loading and maintaining multi-valued indexes in a manner that is transactionally compliant but still efficient.
(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.
(5) Two sets of experiments that evaluate the efficacy of such indexes for applicable queries and how our implementation fares against those in two other document databases: MongoDB and Couchbase Query.

The rest of this paper is structured as follows: Section 2 details related work around multi-valued indexing. Section 3 reviews 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 and Section 7 evaluate the performance of such indexes. Section 8 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 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.

Suppose we want to index vehicles by the names of their vehicle manufacturers. More specifically, we want to index the `Name` attribute inside the `Manufacturer` object reference of a `Vehicle` object. Bertino and Kim studied three approaches to nested indexes in object databases [4]: (i) nested indexes (which map the `Name` attribute to the `Vehicle` objects), (ii) path indexes (which map the same `Name` attribute to both `Manufacturer` and `Vehicle` objects), and (iii) multi-indexes (which first map the `Name` attribute to the `Manufacturer` objects, then map the `Manufacturer` objects to the `Vehicle` objects). Under a relational lens, multi-indexes (item (iii)) can be viewed as pair-wise join indexes, which have also been studied by Valduriez [26]. 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 [16]. As an example, suppose we now want to index all `Name` fields associated with all `Division` objects

© Copyright 2022 for this paper by its author(s). Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

within the `Divisions` list of a `Manufacturer` object. This 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 [11, 17, 20, 21]. Goczyla proposed an extension to set indexing in object databases that not only handles set membership, but the more general cases of superset, subset, and set equality [14].

## 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 [23]. 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 [10], MongoDB [22], and Oracle's NoSQL database [24] 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" [12], 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 undesirable ambiguities for the user (in terms of structure and needless repetition, as we will discuss later).

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 [7, 19]. Relational databases with document extensions like MySQL [9] and PostgreSQL [25] 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 ASTERIXDB BACKGROUND

In this section we give an overview of AsterixDB and single-valued B+ tree indexes in AsterixDB.

## 3.1 System Overview

AsterixDB is a big data management system (BDMS) designed to be a highly scalable platform for information storage, search, and analytics [2]. To scale outward it follows a shared-nothing architecture, where each node independently accesses storage and memory. All nodes 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 translated into a logical plan and subsequently given to a rule-based optimizer to produce an optimized logical plan [5]. This optimized logical plan is then translated into a job that can run across all nodes in the cluster [6]. 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 [1].

## 3.2 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`. Suppose that we have a dataset `Users` and we want to create a composite secondary B+ tree index on two string fields: the `name` field and the `zip_code` field inside of an object field `address`. We would then issue the statement in Listing 1 to AsterixDB:

```
1 CREATE INDEX userNameZipIdx ON Users (
2     name : string,
3     address.zip_code : string
4 );
```

**Listing 1: Syntax for creating a single-valued composite secondary B+ tree index in AsterixDB.**

The index creation statement in Listing 1 is composed of three 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. All paths are followed by their endpoint data type, in this case '`: string`', though this part is excluded if the fields are defined in the `Users` type definition. 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. Once an AsterixDB user issues the index creation statement in Listing 1, 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 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

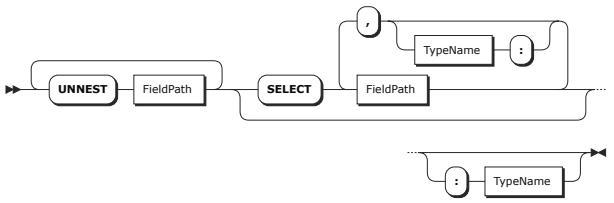


**Figure 1: Syntax for a multi-valued element in a `CREATE INDEX` statement. Due to space constraints, this diagram has been split into two (joined by the three dashes) and the `FieldPath` production refers to either a singular field *or* a dot separated list of fields (denoting nested fields).**

we should not complicate the syntax by allowing general expressions to be indexed.

(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 Figure 1.

We demonstrate several examples in Listing 3 for the datasets described in Listing 2, some of which will also serve to guide discussion in the following section. The first statement in Listing 3 indexes the categories associated with a store, where a category value is a string 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 string values inside 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 4 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 represented by a string field within 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 multi-valued fields is nearly identical to the specification for composite single-valued indexes.

```
// Sample document for Stores.
{
  "store_id": "A35D3",
  "name": "Cat's Kitchen"
  "categories": [
    "Produce",
    "Pet Food"
  ]
}

// Sample document for Users.
{
  "user_id": "34SF4",
  "name": "Mary",
  "address": {
    "street": "3133 Park Place",
    "zip_code": "14622"
  },
  "phones": [
    { "kind": "mobile",
      "number": "808-123-4456" },
    { "kind": "office",
      "number": "555-234-1235" }
  ]
}

// Sample document for Orders.
{
  "order_id": "A35DA",
  "user_id": "3SSS2",
  "store_id": "33378",
  "orderline": [
    { "item_id": 12221,
      "tags": [ "341DD", "2225F" ] },
    { "item_id": 15321,
      "delivery_d": "2021-01-20" }
  ]
}
```

**Listing 2: Sample documents of the datasets `Stores`, `Users`, and `Orders` indexed in in Listing 3 and Listing 4.**

```
CREATE INDEX storesCatIdx ON Stores (
    UNNEST categories : string
);

CREATE INDEX ordersItemIDIdx ON Orders (
    UNNEST orderline
    SELECT item_id : bigint
);

CREATE INDEX ordersTagIdx ON Orders (
    UNNEST orderline
    UNNEST tags : string
);
```

**Listing 3: Example specification for three multi-valued indexes.**

```
CREATE INDEX userNumberKindIdx ON Users (
    UNNEST phones
    SELECT number : string,
           kind : string
);

CREATE INDEX userNameNumberIdx ON Users (
    name : string,
    UNNEST phones
    SELECT number : string
);
```

**Listing 4: Example specification for two composite multi-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 5, the two resulting index keys for this single-valued index would be <`"Raspberry Store"`, `"A34AD"`> and <`"Raspberry Store"`, `"1939D"`>.

Leveraging the layered architecture of AsterixDB, an index entry in a multi-valued B+ tree index is really no different 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 5: <`"Produce"`, `"A34AD"`>, <`"Snacks"`, `"A34AD"`>, and *two* instances of <`"Hardware"`, `"1939D"`>. This non-uniqueness leads to several issues, the most notable being concurrency (discussed in Subsection 5.3). 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.

```
1 { "store_id": "A34AD",
2   "name": "Raspberry Store",
3   "categories": ["Produce",  "Snacks"] }
4 { "store_id": "1939D",
5   "name": "Raspberry Store",
6   "categories": ["Hardware", "Hardware"] }
```

**Listing 5: Two sample documents from the `Stores` dataset.**

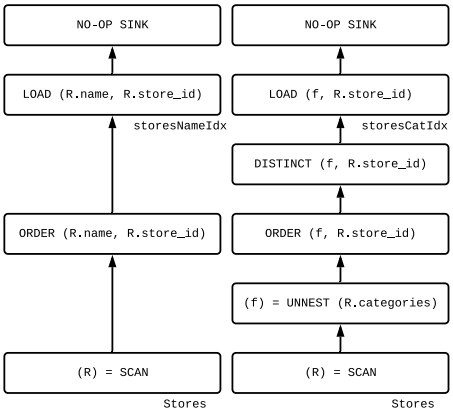


**Figure 2: Two data flows for bulk-loading an index. The left one is the data flow for bulk-loading a single-valued index, while the right one is the data flow for bulk-loading a multi-valued index.**

## 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 [13]). Two data flows for the creation of a secondary index are illustrated in Figure 2. 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 Figure 2 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).

Before diving into the 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 [18]. 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. Similar to bulk-loading, the principles explored here also extend to the operations not discussed (i.e. `DELETE` and `UPSERT`, for details see [13]). 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 Figure 3. The left describes the data flow for performing 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 to us 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 Figure 3, 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 (see [13] for an explanation of why a subplan-based data flow is advantageous here). A subplan is an isolated DAG of operators that is used by the containing operator to perform some computation on a value and then utilize the DAG's output for the

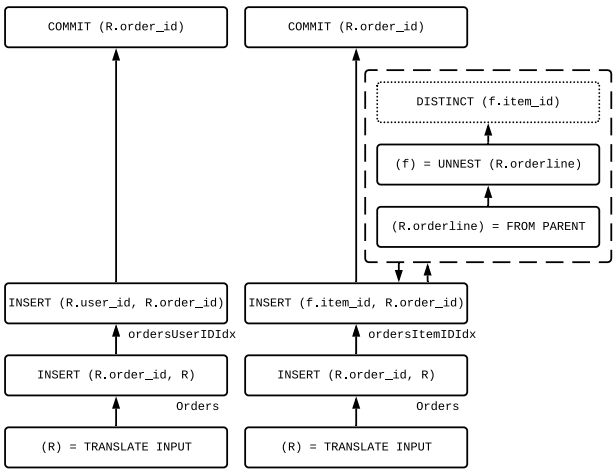


**Figure 3: Two separate data flows for performing an `INSERT` statement on a dataset. The left describes the data flow for executing an `INSERT` on a dataset with one single-valued index, while the right describes the data flow for executing an `INSERT` on a dataset with one multi-valued index.**

containing operator's computation. For this particular subplan and containing operator `INSERT`, the value given to the DAG is the `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 dotted lines composing the node in the figure) is performed by the storage layer via duplicate key rejection.

## 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 6 describes an existential quantification query that aims to find all users that have an office phone.

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

**Listing 6: A SQL++ existential quantification query.**

If an applicable index exists (i.e., a multi-valued index on the `name` field inside the `phones` array of objects), then the query plan in Figure 4 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

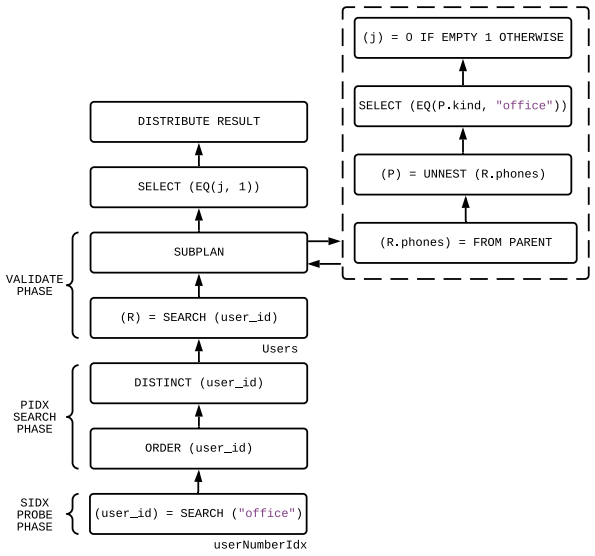


**Figure 4: Index leveraging plan to execute the existential quantification query in Listing 6.**

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 [15] (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 `"mobile"` 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 Figure 4 can also be extended to queries with a join that requires the values of a multi-valued field. Listing 7 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 Figure 5 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

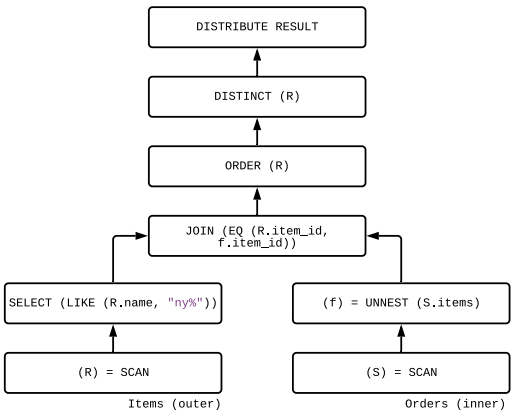


**Figure 5: Non-index-leveraging query plan to execute the array-probing join query in Listing 7.**

```
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 7: A join query that probes the inside of an multi-valued field of another dataset.**

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 7. The AsterixDB optimizer would recognize this and generate the plan illustrated in Figure 6, which logically performs an index-nested loop join (INLJ). We divide the join using the same three phases from Figure 4: 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.

Having described why the plans in Figure 4 and Figure 6 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 Figure 5 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

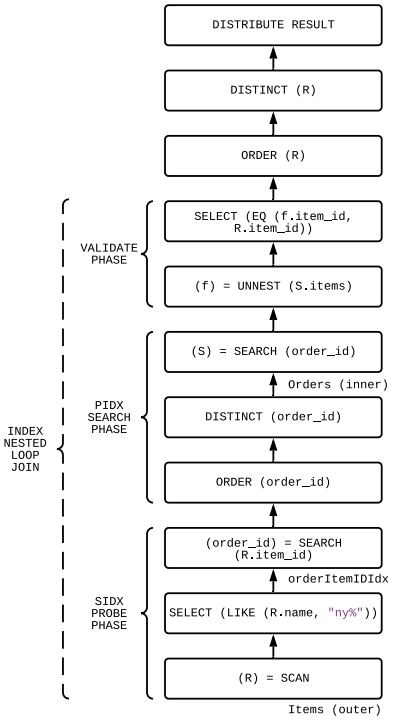


**Figure 6: Index-leveraging query plan to execute the array-probing join query in Listing 7.**

match the structure defined in $i$. Given the `orderItemIDIdx` index specification in Listing 3, 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 the Figure 5 example, the `SIDX_PROBE` subgraph is composed of the outer branch (`Items SCAN` and the original `SELECT` operator for `R.name`), and a $\texttt{SEARCH}_{\texttt{SIDX}}$ with the relevant conjuncts $C$. The `PIDX_PROBE` subgraph is composed of the `ORDER`, `DISTINCT`, and the $\texttt{SEARCH}_{\texttt{PIDX}}$ 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 7 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

**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` := ($\texttt{SEARCH}_{\texttt{SIDX}}(C)$);
    `PIDX_SEARCH` := (`ORDER(PK)` ⟶ `DISTINCT(PK)` ⟶ $\texttt{SEARCH}_{\texttt{PIDX}}$(`PK`));
    `VALIDATE` := (`op`);
    replace (`op_S`) in $Q$ with (`SIDX_PROBE` ⟶ `PIDX_SEARCH` ⟶ `VALIDATE`);
  **end**
  **else**
    `SIDX_PROBE` := (outer branch of `op` ⟶ $\texttt{SEARCH}_{\texttt{SIDX}}(C)$);
    `PIDX_PROBE` := (`ORDER(PK)` ⟶ `DISTINCT(PK)` ⟶ $\texttt{SEARCH}_{\texttt{PIDX}}$(`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**;

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 Equation 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 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 $U$.

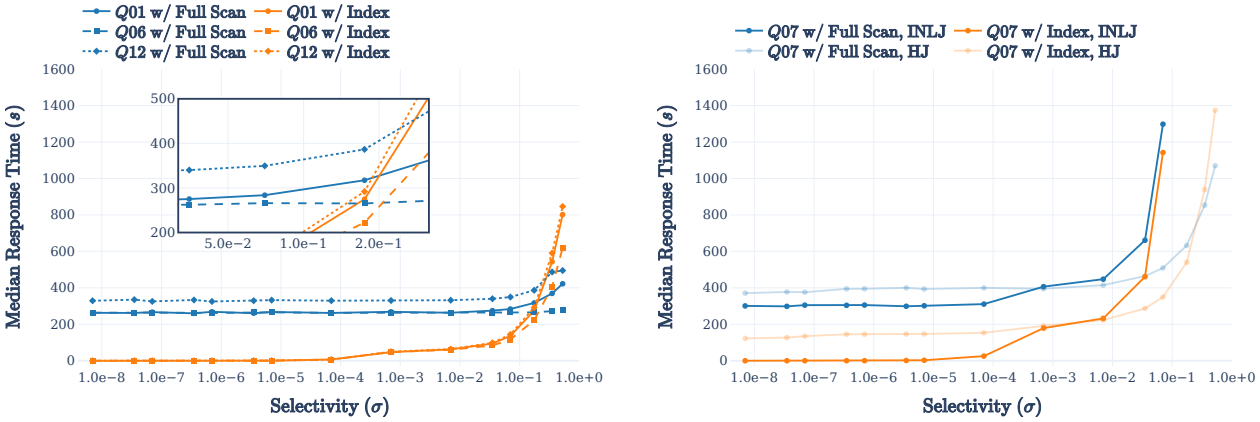


**Figure 7: Median response time vs. selectivity for several queries on an AsterixDB instance. We compare executions that utilize an multi-valued index, perform a full scan of the data, use INLJ to perform joins, and use hybrid-hash to perform joins.**

## 6 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.

### 6.1 Experimental Setup

All experimental runs were performed on a single-node instance of AsterixDB, executed on an AWS c5.xlarge node, 4 vCPUs @ 3.4GHz with 8GB of RAM and AWS gp2 SSDs. The benchmark queries used here are from CH2 [8], a document-oriented combination of TPC-C and TPC-H 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` (25GB), `Stock` (25GB), and `Customer` (12GB). 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 queries used in this experiment can be found at `https://github.com/glennga/aconitum`.

### 6.2 Results and Analysis

Figure 7 depicts several CH2 queries executed using query plans with a multi-valued index 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 $\sigma$) on the logarithmic x-axis. $\sigma$ represents the fraction of the dataset for which the predicate holds true. Beginning with the left of Figure 7, 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 minutes (a 350x speedup minimum). As $\sigma$ 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 [15].

```
1  FROM     Orders O, Stock S, Customer C,
2           Supplier SU, Nation N1, Nation N2
3  UNNEST   O.orderline OL
4  LET      suppkey = ..., nationkey = ...
5  WHERE    S.w_id = OL.supply_w_id AND
6           S.i_id = OL.i_id AND
7           C.id   = O.c_id AND
8           C.w_id = O.w_id AND
9           C.d_id = O.d_id AND
10          SU.suppkey = suppkey AND
11          N1.nationkey = SU.nationkey AND
12          N2.nationkey = nationkey AND ...;
```

**Listing 8: Portion of query CH2 query 7 (`FROM` and part of `WHERE`), expressed in SQL++.**

The right graph in Figure 7 tells a similar story with a multi-join query, query 7 (partially given in Listing 8 to highlight the datasets being joined). 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 $\sigma$ 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 $\sigma$ 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

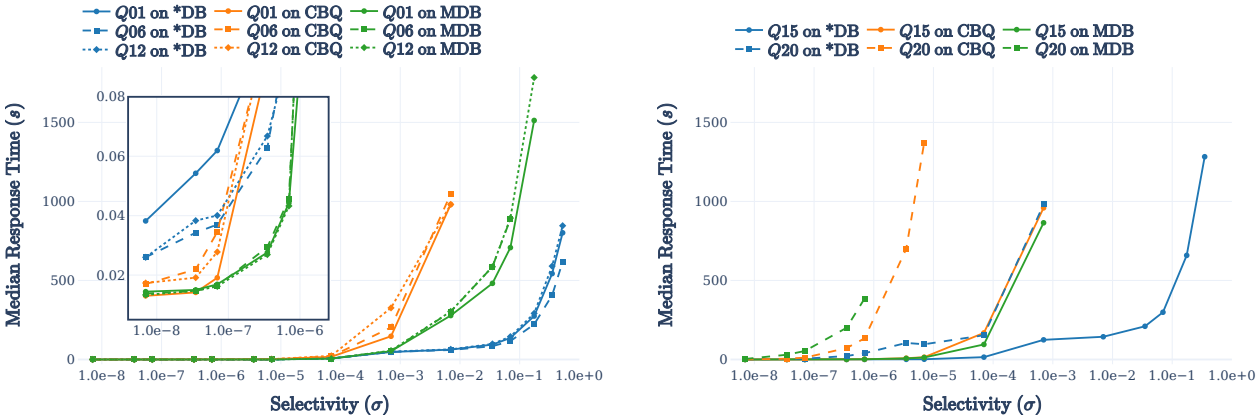


**Figure 8: Median response time vs. selectivity for two sets of queries across an AsterixDB instance (denoted as *DB), a MongoDB instance (denoted as MDB), and the query service on a Couchbase instance (denoted as CBQ).**

```
1  FROM       Orders O
2  UNNEST     O.orderline OL
3  WHERE      OL.delivery_d BETWEEN $1 AND $2
4  GROUP BY   OL.number
5  SELECT     OL.number,
6             SUM(OL.quantity) AS sum_qty,
7             SUM(OL.amount) AS sum_amount,
8             AVG(OL.quantity) AS avg_qty,
9             AVG(OL.amount) AS avg_amount,
10            COUNT(*) AS count_order
11 ORDER BY   OL.number;
```

**Listing 9: CH2 query 1, expressed in SQL++ (and also N1QL for Query, given the similarities in each query language). Selectivity is varied by modifying variables '$1' and '$2'.**

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 $\sigma$ 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 $\sigma$ 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.

## 7 SYSTEM COMPARISON EVALUATION

In this section, we compare the performance of queries that can benefit from multi-valued indexes running on AsterixDB, MongoDB, and Couchbase Query.

### 7.1 Experimental Setup

All experimental runs were performed on single-node instances of AsterixDB, MongoDB and Couchbase, executed on an AWS c5.xlarge node, 4 vCPUs @ 3.4GHz with 8GB 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 minutes 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 AsterixDB queries, Couchbase Query queries, and MongoDB aggregation pipelines used in this experiment can be found at `https://github.com/glennga/aconitum`.

### 7.2 Results and Analysis

Figure 8 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 Figure 7). 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 13ms at $\sigma$ = 7.0e-8, but queries 6 and 12 on Couchbase Query run roughly 3ms slower than queries 6 and 12 on MongoDB at the same $\sigma$. At these low selectivities values, AsterixDB executes the slowest with a minimum response time of 25ms. 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, given in Listing 9. 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 $\sigma$. 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 minute 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 Figure 8, 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 $\sigma$? With query 15, both MongoDB and Couchbase Query were able to execute for increasing values of $\sigma$ 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 $\sigma$ values thus far across any system for MongoDB: a maximum of $\sigma$ = 7.0e-7 before executing beyond 30 minutes. 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 minutes.

## 8 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 two sets of experiments that evaluated the efficacy of multi-valued indexes in AsterixDB. We would 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.

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 to accelerate queries that only involve the prefix of the sort key, and (c) storing empty arrays and multisets in order to accelerate general universal quantification queries (i.e. remove the non-emptiness clause).

## 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 NSF 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).

## REFERENCES

[1] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J. Carey, Markus Dreseler, and Chen Li. 2014. Storage Management in AsterixDB. *Proc. VLDB Endow.* 7, 10 (June 2014), 841–852. https://doi.org/10.14778/2732951.2732958

[2] AsterixDB. 2021. Apache AsterixDB, a Scalable Open Source Big Data Management System (BDMS). Available at https://asterixdb.apache.org.

[3] E. Bertino and P. Foscoli. 1995. Index Organizations for Object-Oriented Database Systems. *IEEE Transactions on Knowledge and Data Engineering* 7, 2 (1995), 193–209. https://doi.org/10.1109/69.382292

[4] E. Bertino and W. Kim. 1989. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering* 1, 2 (1989), 196–214. https://doi.org/10.1109/69.87960

[5] Vinayak Borkar, Yingyi Bu, E. Preston Carman, Nicola Onose, Till Westmann, Pouria Pirzadeh, Michael J. Carey, and Vassilis J. Tsotras. 2015. Algebricks: A Data Model-Agnostic Compiler Backend for Big Data Languages. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (Kohala Coast, Hawaii) *(SoCC '15)*. Association for Computing Machinery, New York, NY, USA, 422–433. https://doi.org/10.1145/2806777.2806941

[6] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. 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, 1151–1162. https://doi.org/10.1109/ICDE.2011.5767921

[7] Simran Brucherseifer. 2021. Index Basics. Available at https://www.arangodb.com/docs/stable/indexing-index-basics.html.

[8] M. Carey, D. Lychagin, M. Muralikrishna, V. Sarathy, and T. Westmann. 2021. CH2: A Hybrid Operational/Analytical Processing Benchmark for NoSQL. In *13th TPC Technology Conf. on Performance Evaluation & Benchmarking (TPC TC)* (Copenhagen, Denmark). TPCTC.

[9] Oracle Corporation. 2021. 13.1.15 CREATE INDEX Statement. Available at https://dev.mysql.com/doc/refman/8.0/en/create-index.html.

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

[11] O. Deux. 1990. The Story of O2. *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (1990), 91–108. https://doi.org/10.1109/69.50908

[12] Simon Dew. 2021. Flex Indexes. Available at https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/flex-indexes.html.

[13] Glenn Galvizo. 2021. *On Indexing Multi-Valued Fields in AsterixDB.* Master's thesis. Computer Science Department, University of California, Irvine.

[14] K. Goczyla. 1997. 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, 47–54. https://doi.org/10.1109/EURMIC.1997.617215

[15] Goetz Graefe. 2011. Modern B-Tree Techniques. *Found. Trends Databases* 3, 4 (April 2011), 203–402. https://doi.org/10.1561/1900000028

[16] Alfons Kemper and Guido Moerkotte. 1990. Access Support in Object Bases. *SIGMOD Rec.* 19, 2 (May 1990), 364–374. https://doi.org/10.1145/93605.98745

[17] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. 1990. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (1990), 109–124. https://doi.org/10.1109/69.50909

[18] Young-Seok Kim. 2016. *Transactional and Spatial Query Processing in the Big Data Era.* Ph.D. Dissertation. University of California, Irvine.

[19] Rich Loveland. 2021. Inverted Indexes. Available at https://www.cockroachlabs.com/docs/v20.2/inverted-indexes.

[20] David Maier and Jacob Stein. 1986. Indexing in an Object-Oriented DBMS. In *Proceedings on the 1986 International Workshop on Object-Oriented Database Systems* (Pacific Grove, California, USA) *(OODS '86)*. IEEE Computer Society Press, Washington, DC, USA, 171–182.

[21] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. 1986. Development of an Object-Oriented DBMS. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '86)*. Association for Computing Machinery, New York, NY, USA, 472–482. https://doi.org/10.1145/28697.28746

[22] MongoDB. 2021. Multikey Indexes. Available at https://docs.mongodb.com/manual/core/index-multikey/.

[23] Matthias Nicola and Bert Van der Linden. 2005. Native XML Support in DB2 Universal Database. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi (Eds.). ACM, Trondheim, Norway, 1164–1174. http://www.vldb.org/archives/website/2005/program/paper/thu/p1164-nicola.pdf

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

[25] Teodor Sigaev and Oleg Bartunov. 2008. Gin for PostgreSQL. Available at http://www.sai.msu.su/~megera/wiki/Gin.

[26] Patrick Valduriez. 1987. Join Indices. *ACM Trans. Database Syst.* 12, 2 (June 1987), 218–246. https://doi.org/10.1145/22952.22955