UNIVERSITY OF CALIFORNIA,
IRVINE

On Indexing Multi-Valued Fields in AsterixDB

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTERS OF SCIENCE

in Computer Science

by

Glenn Galvizo

2021

Contents

List of Figures

List of Code Listings

ACKNOWLEDGMENTS

I would like to express my sincerest gratitude to my advisor Dr. Michael Carey for his support, encouragement, direction, and knowledge these past two years. I am excited to work with him more in the years to follow. I would also like to thank my thesis committee: Dr. Chen Li and Dr. Faisal Nawab. Ian Maxon, Chen Luo, and Wail Alkowaileet have my gratitude as well for getting me up to speed with the AsterixDB project and its processes. Vijay Sarathy has my appreciation for helping me with the CH benchmark. Special thanks to Dmitry Lychagin for acting as a code reviewer and for providing input / direction with respect to AsterixDB specifics. Lastly, I would like to thank my family for their love and support.

This research is supported in part by the NSF awards IIS-1838248, IIS-1954962, and CNS-1925610, by the HPI Research Center in Machine Learning and Data Science at UC Irvine, and from the Donald Bren Foundation (via a Bren Chair).

ABSTRACT OF THE THESIS

On Indexing Multi-Valued Fields in AsterixDB

By

Glenn Galvizo

Masters of Science in Computer Science

University of California, Irvine, 2021

Professor Michael J. Carey, Chair

Secondary indexes in database systems are traditionally built under the assumption that one data record maps to one indexed value. Nowadays single data records often hold collections of values that users want to access efficiently in an ad-hoc manner. Database users are thus torn between changing their data model to support such indexes or living with subpar query execution time. 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.

This thesis details 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. A non-ambiguous, clean, and concise syntax is first developed for specifying such indexes. Data flows for bulk-loading a collection of records and maintaining an index correctly with respect to concurrency are then illustrated. Query plans to take advantage of multi-valued indexes for use in joins involving arrays / multisets, predicates with existential quantification, and predicates with universal quantification (the latter of which is not supported by any other system to date) are discussed next. We finally conclude with experiments demonstrating the efficacy of these indexes.

Chapter 1

Introduction

Indexing, the concept of organizing data for more efficient search, is a technique older than computing itself. In the field of database systems, secondary indexing refers to the practice of duplicating data from some primary data source for the specific purpose of efficient retrieval. A single data record can assume many different forms, presented in the following (non-exhaustive) list: (i) as a row in a table, (ii) as a object from a class, (iii) as a key-value pair, (iv) as a node in a graph, or (v) as a self-describing document. The representation in item (v) has grown popular over the years, mainly due to the flexibility of not committing to a data description for all of the records in one's database. Consequently this is the data model utilized by the system that this research has been performed on, *AsterixDB*, a NoSQL-style big data management system boasting a structured query language (SQL++) over a collection of documents.

Secondary indexes come in many different flavors with a rich history of research. We will start our discussion with secondary indexes in a traditional relational database management system. In traditional relational systems (utilizing the representation in item (i) from the previous paragraph), a data record has a single value per field (column). A secondary index on a single field for a collection of records is known as a single-field *atomic* index, while a secondary index on multiple fields is known as a *composite* atomic index. Now suppose that we relax the constraint on our relational system to allow for multiple values per field (such as an array or set) for a single record. If we create a secondary index on such a multi-valued field, we refer to this as a *multi-valued* index. A multi-valued index is distinct from a composite atomic 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 thesis focuses on supporting secondary indexes for multiple values per-record. There exist many different ways to utilize a secondary index for use in accelerating queries, but this research focuses on using secondary indexes in cooperation with the primary data source. This research does not discuss how to utilize multi-valued secondary indexes to act as the *sole* data source for a select few queries. The main contributions of this thesis are two-fold: (i) an analysis of various syntax for specifying indexes on multiple values within a single record, and (ii) novel data flows that utilize said indexes for ad-hoc queries in a structured query language for documents.

The remainder of this thesis is structured as follows: Chapter 2 discusses related work around indexing with multiple endpoints. Chapter 3 provides background into what AsterixDB is and the lifecycle of a query. Chapter 4 describes the syntax for specifying a multi-valued index and the types of queries that a user can expect to be accelerated using said indexes. Chapter 5 details various data flows to realize multi-valued indexing. Chapter 6 evaluates the performance of such indexes. Chapter 7 concludes this thesis and details potential future work with respect to multi-valued indexing.

Chapter 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: (a) indexing in object-oriented databases, and (b) 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

Work in this area dates back approximately 30 years. We discuss the object data model first. Here, objects *and* their member objects are each first class citizens. This contrasts with the relational model, where the attributes of a tuple must always be tied to a tuple itself. In terms of indexing, object-oriented databases must address the problem of what exactly one should index when objects can reside in objects. To motivate the object indexing problem, Figure 2.1 details the following: A `Vehicle` contains two strings (`Model` and `Color`) and one object (`Manufacturer`). A `Moped` inherits from the `Vehicle` object, and thus has

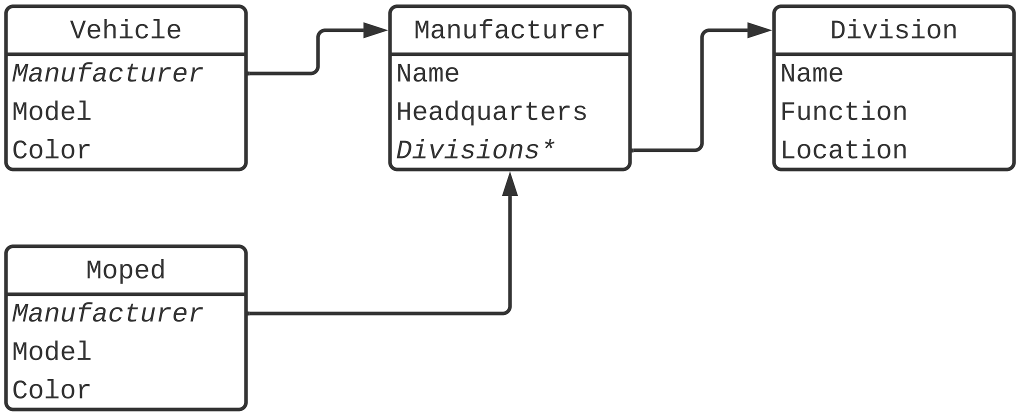


Figure 2.1: Class-attribute hierarchy for a vehicle-manufacturer example.

the same attributes. A `Manufacturer` contains two strings (`Name` and `Headquarters`) and one list of objects (`Divisions`). A `Division` object contains three strings: `Name`, `Function`, and `Location`.

Suppose we want to index vehicles by the names of their vehicle manufacturers. Using the class-attribute hierarchy detailed in Figure 2.1, we specifically want to index the `Name` attribute inside the `Manufacturer` object 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 been studied by Valduriez [29]. 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]. Note that in Figure 2.1, the class `Moped` and `Vehicle` are essentially treated as two completely separate classes. 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 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, 20, 21, 17]. 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 [13].

2.2 Multi-Valued Indexing in Document Databases

Next we address the document model. While not as radical as the object data model, the document model differs from the relational model in that the attributes of a document are not flat (i.e. the inclusion of composite and multi-valued attributes) and that the documents themselves are self-describing (lending itself to weaker type assumptions). Take the XML document, where an element is composed of many sub-elements and there exists no way to determine if a sub-element will be atomic or multi-valued. The XML extension for DB2 addresses the atomic vs. multi-valued problem with respect to indexing by treating every element as a potential multi-valued attribute [24]. The JSON document, in contrast to the XML document, does allow one to specify if a field is multi-valued or not (making the atomic vs. multi-valued problem a non-issue). Modern JSON document stores such as Couchbase [10], MongoDB [22], and Oracle's NoSQL database [25] have support for multi-valued indexing, but all had somewhat different design goals than the multi-valued indexing approach studied here. Couchbase's array indexes are made with the intent to *cover* certain queries (i.e., to only use the index to satisfy a query), while AsterixDB's multi-valued indexes were designed to handle a larger set of queries at the cost of no longer being covering. Couchbase 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 queries that can be answered using a B+ tree. Finally, MongoDB and Oracle's index specification syntax leave undesired ambiguity for the user (as we will discuss later).

The document model is not restricted to just document databases. This model is also available 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 [27] also support a limited form of multi-valued indexing, but again only support membership queries. Multi-valued indexes in AsterixDB on the other hand support a much larger set of queries, such as joins with a value inside a multi-valued field, existential quantification, and universal quantification (the latter of which is not supported by any of the systems researched here to date).

Chapter 3

Overview of AsterixDB

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. The general system architecture for AsterixDB is given in Figure 3.1. To describe AsterixDB, we detail the lifecycle of a general request (e.g. an insertion, update, query, etc. . .) below.

1. All requests first arrive at the client interface on the *cluster controller*. This cluster controller component not only serves as an entry point for all user requests, but also coordinates all work amongst the individual AsterixDB nodes.
2. The request is then given to the *SQL++ compiler*. Here our request is translated into a logical plan and subsequently given to a rule-based optimizer to produce an optimized logical plan [5].
3. To determine the different access paths, the SQL++ compiler must also be able to access various metadata about the data that AsterixDB is managing. This includes what datasets and indexes are available. The *Metadata Manager* relays this infor-

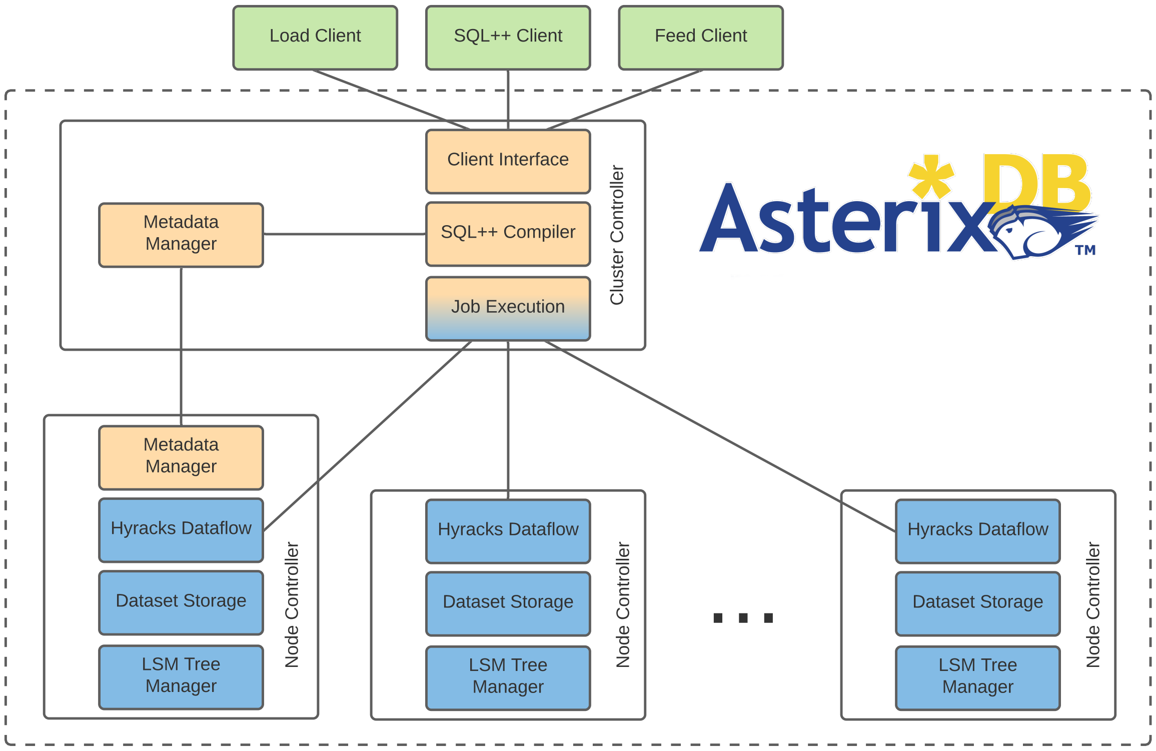


Figure 3.1: Overview of the AsterixDB system.

mation between the node containing the aforementioned metadata and the SQL++ compiler.

4. The logical plan produced by the SQL++ compiler is then given to the *Job Execution* component, which will translate the plan into a job that can be executed across all nodes in the cluster [6]. Datasets in AsterixDB are partitioned across the cluster on their primary key into primary indexes, where the data records reside, with all secondary indexes being local to each node. With this partitioning in mind, the appropriate jobs are submitted to each node's *Hyracks Dataflow* component.
5. With each node now having its own job, it executes the job and interacts with its disk through the *Dataset Storage* and *LSM Tree Manager* components [1]. All datasets and indexes in AsterixDB use LSM (log-structured merge) trees for their native storage.
6. Once a job finishes with the *Job Execution* components, its status and any query results are relayed back to the client.

The work in this thesis primarily focuses on the *SQL++ Compiler* and *Job Execution* components of the cluster controller and the *Hyracks Dataflow* component of a node controller.

Chapter 4

Indexing User Experience

Ideally, a user should only have to interact with an index once: when the user creates the index. By issuing applicable queries, the user should then experience a faster response time. Hence this chapter is divided into three sections: (i) a example database to guide the following discussion, (ii) how a user should specify the creation of a multi-valued index, and (iii) what types of user queries can be accelerated through the use of a multi-valued index.

4.1 Inventory Management Example

To illustrate many of the topics mentioned in this chapter and the next, an inventory management example (dubbed `ShopALot`) is presented. There are four datasets associated with this example: (i) `Users`, who place (ii) `Orders` from (iii) `Stores` that sell (iv) `Products`.

4.1.1 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 0 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 4.1: Type definition for the Users dataset.

4.1.2 Orders Dataset

The Orders dataset represents orders placed by users to some store. An order is uniquely identified by an order_id and has a one-to-many relationship 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 selling_price, a one-to-many relationship with Products (represented as product_id), and an array tags.

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

Listing 4.2: Type definition for the Orders dataset.

4.1.3 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 4.3: Type definition for the Stores dataset.

4.1.4 Products Dataset

The Products dataset represents products sold by stores. A product is uniquely identified by a product_id and possesses a single category, name, and description.

```
CREATE TYPE ProductsType AS {
    product_id: string,
    category: string,
    name: string,
    description: string
};
```

Listing 4.4: Type definition for the Products dataset.

4.2 Creating an Index

To set the scene, we first describe how indexes are currently created in AsterixDB. To create an index, a user can use the (simplified) syntax specified below:

```
CREATE INDEX idxName ON DatasetName (
    element1, element2, element3,  ...
);
```

Listing 4.5: General syntax for creating an index in AsterixDB.

In the example above, element1, element2, element3, etc... represent *elements* of the dataset with name DatasetName that are contained in this index. An index element here refers to either (i) a field, (ii) a sequence of fields, or (iii) an expression that generates one or more fields. The following sections will describe and motivate the issues associated with describing a multi-valued element in AsterixDB. We will then conclude with the new syntax for multi-valued index specification.

4.2.1 Index Specification Without Non-1NF Features

An AsterixDB `CREATE INDEX` statement is given below for a composite index on the `category` and `name` fields of the `Products` dataset.

```
CREATE INDEX productsCatNameIdx ON Products (
    category, name
);
```

Listing 4.6: Specification for a composite index on two fields.

The `Products` dataset is in first normal form (1NF), possessing no composite or multi-valued attributes. Had `Products` existed in a relational system, the `CREATE INDEX` statement for this dataset would be nearly identical. In Listing 4.6, index fields are indistinguishable from the abstraction given at the start of this section: *elements*. To accommodate the aforementioned non-1NF features requires us to go beyond just fields.

4.2.2 Index Specification With Nested Fields

Another AsterixDB `CREATE INDEX` statement is given for two index elements (the `first` and `last` fields of the `name` object field) associated with the `Users` dataset.

```
CREATE INDEX usersNameIdx ON Users (
    name.first, name.last
);
```

Listing 4.7: Specification for a composite index on two *nested* fields.

In contrast with the previous example (where only two fields exist), a total of three fields are required to describe two index elements: `first`, `last`, and `name`. The coupling of `name` and `first`, and `name` and `last`, are henceforth denoted as *paths*. More generally, a *path* is an ordered sequence of fields. Using a "`.`" to denote that the field to the right of the dot is found within the field to the left has found widespread use in systems that allow fields within fields. Continuing with the example above, the path "`name.first`" describes the `first` field inside the `name` object.

4.2.3 Index Specification With Multi-Valued Fields

Paths allow us to cleanly describe nested fields, but complications arise when one field refers to many values (i.e. the concept of multi-valuedness). Listed below are several candidate approaches to index specification for fields with multi-valuedness. Each will propose a solution and motivate an issue to be answered by the next candidate, culminating in the final syntax for multi-valued elements in AsterixDB.

Candidate 1: Not Distinguishing Between Multi-Valued and Atomic Fields

The example described in Listing 4.7 applies to atomic fields only, but let us assume this restriction was absent. The following hypothetical `CREATE INDEX` statement could then be invoked to create an index on a field within an array. The specification below could describe an index on the `number` field inside the array of `phones` objects for the `Users` dataset:

```
CREATE INDEX usersNumberIdx ON Users (
    phones.number
);
```

Listing 4.8: Candidate 1 example specification for multi-valued elements.

If the type of the field `phones` is known a priori, it is easy to infer that the `number` field is located not within an object, but within an array of objects. This however is too strong of an assumption to make with document databases. Had the type for `phones` not been known ahead of time, it would be impossible to tell if the path "`phones.number`" is associated with one value per record or multiple values per record without first fetching a record.

MongoDB adopts the approach of treating atomic and multi-valued fields as being the same (with respect to index specification), deferring the interpretation of the index path to index-maintenance time. To show some of the anomalies associated with this approach, suppose that we have two documents:

```
{ "user_id": "A1", "phones": { "number": "123-4567" } }
{ "user_id": "A2", "phones": [ { "number": "123-4567" } ] }
```

Listing 4.9: Example documents for the `Users` dataset. The type for the `phones` field is not strictly enforced.

Given the path "`phones.number`", MongoDB would choose to store both entries in the index. In the context of MongoDB's query language (which allows the same ambiguities) this syntax is permissible, but SQL++ is more precise with respect to the structure of applicable results. If the MongoDB approach were taken for AsterixDB, an additional check for the field's structure would need to be applied to remove the index entries that do not structurally match the query. As an example, suppose the `usersNumberIdx` index contains all documents listed above. Now assume that the optimizer chooses to use this index as the access path for the query below.

```
FROM     Users U
UNNEST   U.phones P
WHERE    P.number = "123-4567"
SELECT   DISTINCT VALUE U.user_id;
```

Listing 4.10: SQL++ query for implicit existential quantification on an array field.

The correct answer to this query is `["A2"]`, but if the index does not store per-record structure information for the query to utilize (or if the records themselves are not fetched after the index lookup and a structural check is applied for each index entry), then the result will incorrectly become `["A1", "A2"]`. This situation is illustrated in Figure 4.1. Thus candidate 1 illustrates the first design requirement for our new index specification syntax: clearly distinguish between atomic and multi-valued fields.

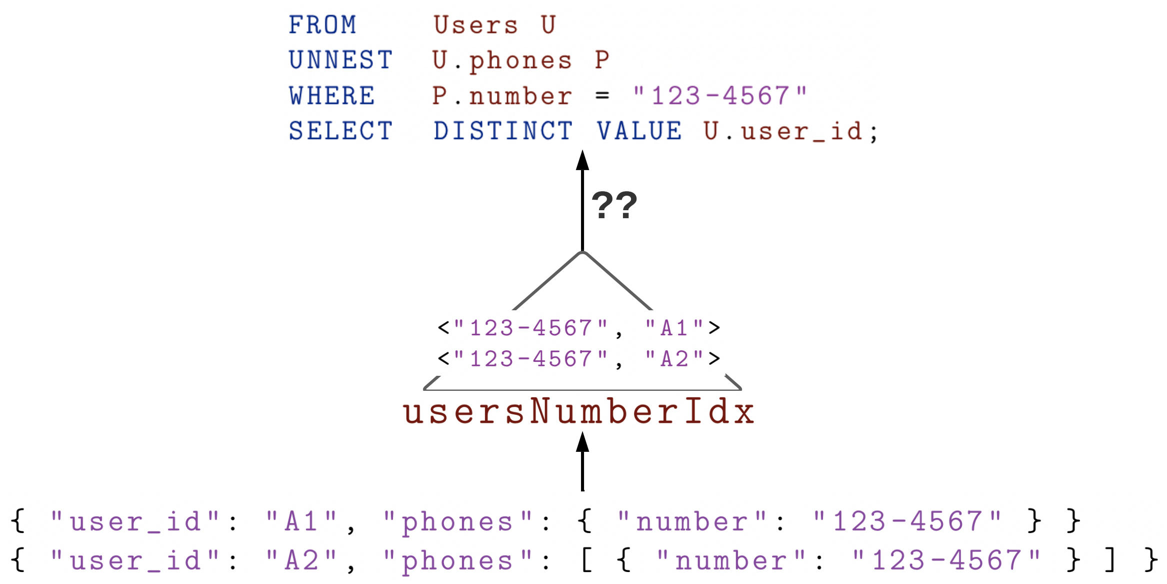

FROM Users U
UNNEST U.phones P
WHERE P.number = "123-4567"
SELECT DISTINCT VALUE U.user_id;
??
<"123-4567", "A1">
<"123-4567", "A2">
usersNumberIdx
{ "user_id": "A1", "phones": { "number": "123-4567" } }
{ "user_id": "A2", "phones": [{ "number": "123-4567" }] }

Figure 4.1: Overview of the structure information loss situation with candidate approach 1.

Candidate 2: Using an Operator for Multi-Valued Steps

Having described the issues associated with not discriminating between atomic fields and multi-valued fields in paths, we now move to different ways to syntactically denote multi-valued nodes in paths. Specifically we need to be able to express a way to represent objects within an array or set. Amazon's PartiQL, another SQL-style query language for semi-structured data, has a special operator for expressing *all* objects or values inside a list: the wildcard (`[*]`) operator [26]. This wildcard operator in PartiQL is functionally equivalent to the `UNNEST` operator in SQL++. Revisiting Listing 4.8, we could express the `usersNumberIdx` index in a new wildcard-based candidate notation:

```
CREATE INDEX usersNumberIdx ON Users (
    phones[*].number
);
```

Listing 4.11: Candidate 2 example specification for multi-valued elements.

```
{
  "user_id": "B1",
  "phones": [
    { "kind": "MOBILE", "number": "123-456-7890" },
    { "kind": "OFFICE", "number": "000-123-4567" }
  ]
}
```

Listing 4.12: Example document for the Users dataset.

The notation used above always stores number field values in a phones array, preventing the issues associated with candidate 1. Oracle's NoSQL database adopts this approach of using a separate operator to denote multi-valued steps in their index specifications. This syntax itself is not immune to problems though, suppose now we want to create a composite index on the number and kind fields inside the array of phones objects for the Users dataset:

```
CREATE INDEX usersNumberKindIdx ON Users (
    phones[*].number,
    phones[*].kind
);
```

Listing 4.13: Second candidate 2 example specification for multi-valued elements.

To specify a composite index for two fields in the same array for Listing 4.13, the wildcard operator is listed twice for the same phones array (even though phones is only UNNESTed once). Grammar-wise, the first index element phones[*].number is disjoint from the second index element phones[*].kind. This means that a user is grammatically free to specify different arrays / sets for each index element, leading to more complex specifications for multi-valued indexes. MySQL interprets the specification of multiple arrays in their indexing statement as a cross-product, citing the explosion of index entries as a reason to not support such indexes [23]. Additionally, the use cases for a composite index on multiple multi-valued fields are very narrow. Using the cross-product interpretation of such a specification, such an index could only be used when the two multi-valued fields are given in a query that itself performs a cross-product between two such fields. Given that all of the other systems researched in this thesis disallow the creation of indexes on more than one multi-valued field, the design decision was made to not support such a feature as well.

Having eliminated the possibility of indexing fields across more than one multi-valued node, candidate 2 leaves us with syntactically misleading specifications. Suppose the document in Listing 4.12 exists in the `Users` dataset. Listing 4.13 could then be interpreted in two ways:

1. The object holding the `number` and `kind` fields are to be extracted from the `phones` array and indexed *together* (from the same source object). This results in two index entries. (Note that in AsterixDB an index entry consists of all secondary index field values followed by the document's unique identifier, the primary key.)
   (a) `("123-456-7890", "MOBILE", "B1")`
   (b) `("000-123-4567", "OFFICE", "B1")`
2. The `number` field and the `phones` field are to be extracted separately, combined via a cross product, and then indexed. This results in *four* index entries:
   (a) `("123-456-7890", "MOBILE", "B1")`
   (b) `("123-456-7890", "OFFICE", "B1")`
   (c) `("000-123-4567", "OFFICE", "B1")`
   (d) `("000-123-4567", "MOBILE", "B1")`

This issue of ambiguous notation for composite multi-valued indexes (for more than one field within a multi-valued field) illustrates the second design requirement for our index specification syntax: avoid syntactic repetition when specifying multi-valued fields.

Candidate 3: Index Specification using Queries

All the approaches described thus far deal with using special data-definition language (DDL) notations for *paths*, begging the question: "Why not specify indexes using a subset of the query language itself?". This is the approach adopted by Couchbase's array indexes, and is detailed in Listing 4.14.

```
CREATE INDEX usersNumberKindIdx ON Users (
    DISTINCT ARRAY [p.number, p.kind]
    FOR p IN phones END
);
```

Listing 4.14: Candidate 3 example specification for multi-valued elements.

While candidate 3 satisfies the first and second design requirements of our index specification syntax, it is arguably too verbose for our needs (making the `CREATE INDEX` statement hard to read). Couchbase's array indexes were designed with the intent to also act as *covering* indexes. Consequently, Couchbase allows users to specify whether or not all elements of the array should be indexed (vs. just unique elements) and to quantify the array elements to be indexed using an optional `WHEN` clause. Multi-valued indexes in AsterixDB are designed only to *accelerate* many queries, not act as the sole access path for a select few queries. Having been reminded of the design decision to not cover queries, candidate 3 illustrates the third and fourth requirements for our index specification syntax: 3) constrain the specification language for creating non-covering indexes and 4) make the specification easy to read and understand.

4.2.4 Creating a Multi-Valued Index in AsterixDB

We now discuss the new index creation syntax for multi-valued indexes in AsterixDB. To recap the previous sections, this syntax was designed with the following requirements in mind:

1. Distinguish between atomic and multi-valued fields.
2. Avoid repetition when specifying the multi-valued field in composite indexes.
3. Constrain the specification language for creating non-covering indexes.

4. Have an easy-to-read index specification.

To satisfy both requirements 3 and 4 while building upon candidate specification 3, the index DDL used in AsterixDB only allows two additional operators: `UNNEST` and `SELECT`. Revisiting the `usersNumberKindIdx` example, the DDL to create an index on two fields within one multi-valued field is given in Listing 4.15.

```
CREATE INDEX usersNumberKindIdx ON Users (
    UNNEST phones SELECT number, kind
);
```

Listing 4.15: Example specification for a multi-valued element in AsterixDB.

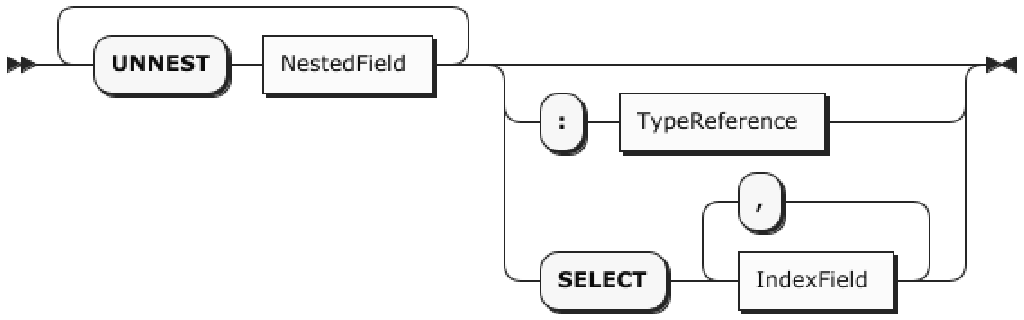
In addition to being concise, this syntax is also easy to debug. A user can transform the previous index specification into a query that can be executed:

```
FROM    Users U
UNNEST  U.phones P
SELECT  P.number, P.kind
```

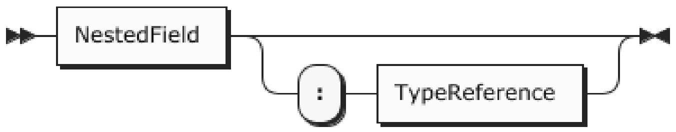
Listing 4.16: Query that corresponds to the index specification in Listing 4.15.

Aside from the inclusion of the `FROM` clause, the only difference between the query in Listing 4.16 and the index specification in Listing 4.15 is Listing 4.16's use of the variable `P`.

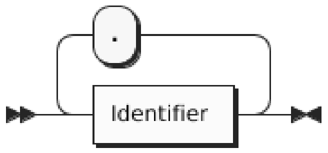
The more general form for a multi-valued index element specification is given in Figure 4.2a with supporting syntax diagrams in Figure 4.2b and Figure 4.2c. A multi-valued index element will always start with an `UNNEST` clause, where `NestedField` represents a *path* to some multi-valued field. If there are many multi-valued elements to traverse to get to our defined endpoint multi-valued element, then `UNNEST` is specified multiple times. If the endpoint multi-valued element holds simple built-in types (such as integers, strings, floats, etc...), then the full index element specification ends there. Otherwise (if the index endpoint(s) are located within objects), `SELECT` must be specified once, followed by all fields or paths to the index endpoint(s). Last but not least, similar to how existing atomic indexes work in



(a) Syntax diagram for MultivaluedIndexElement.



(b) Syntax diagram for IndexField.



(c) Syntax diagram for NestedField.

Figure 4.2: Syntax diagrams for a multi-valued index element.

AsterixDB, a user can also specify the type associated with the to-be-indexed field if the type itself is not specified in the dataset's type definition.

We will now discuss more complex examples using the new index specification syntax.

Example 1: Indexing an Array with Type Specifications

To start, we extend Listing 4.15 to include type specifications for the endpoint fields. If the dataset type for Users did not specify a type for the phones field or if the kind and number field types were not specified in the phones object array, the following CREATE INDEX statement would be used:

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

Listing 4.17: Example specification for a multi-valued element that includes explicit types.

Example 2: Indexing an Array of Built-in Types

Now suppose that we want to create an index on the `categories` array inside the `Stores` dataset. Unlike the previous examples, `categories` is an array of strings (in contrast to an array of objects). To create this index on `categories`, a user would invoke the following `CREATE INDEX` statement:

```
CREATE INDEX storeCatIdx ON Stores (
    UNNEST categories
);
```

Listing 4.18: Example index specification for an array of strings.

Here the `SELECT` clause is absent because there are no fields within the `categories` array to reference.

Example 3: Indexing an Atomic Field and an Array Field

Composite multi-valued indexes are not limited to index endpoints located within a multi-valued field. By only changing how one specifies a *singular* index element (in contrast to many elements), the inclusion of an atomic element outside of the multi-valued field requires no changes to the overall `CREATE INDEX` syntax. The examples below illustrate how a user would create an index on the `Stores` dataset for both the `categories` array and the `zip_code` field of the `address` object field.

```
CREATE INDEX storeZipCatIdx ON Stores (
    address.zip_code,
    UNNEST categories
);
```

Listing 4.19: Example composite index specification with an atomic element prefix and a multi-valued element suffix.

```
CREATE INDEX storeZipCatIdx ON Stores (
    UNNEST categories,
    address.zip_code
);
```

Listing 4.20: Example composite index specification with a multi-valued element prefix and a atomic element suffix.

It is important to note that, as of this writing, the two examples above are not currently implemented in AsterixDB. Listing 4.19 and Listing 4.20 are instead meant to illustrate the future user-level design of composite indexes with atomic and multi-valued elements.

Example 4: Indexing an Array Within an Array

In some cases, a field to be indexed is nested within more than one level of multi-valued field. To demonstrate indexing a field within two array fields (the `tags` field within the `items` object array of the `Orders` dataset), the example below is given:

```
CREATE INDEX ordersItemTagsIdx ON Orders (
    UNNEST items UNNEST tags
);
```

Listing 4.21: Example index specification for an array within an array.

Again this specification can easily be debugged by transforming it into a query and adding aliases:

```
FROM    Orders O
UNNEST  O.items I
UNNEST  I.tags T
SELECT  T
```

Listing 4.22: Query that corresponds to the index specification in Listing 4.21.

4.3 Accelerating Queries

As previously mentioned, a multi-valued index is aimed to accelerate various forms of quantification within some array or multiset. This section aims to describe specific example queries that the optimizer will recognize as applicable for multi-valued index usage. For details regarding *how* the optimizer recognizes applicability, refer to Section 5.5.

Currently there are three types of indexes in AsterixDB: (i) B+ tree indexes, (ii) R tree indexes, and (iii) inverted (text) indexes. All work in this thesis has been performed by extending B+ tree indexes to handle multi-valued fields. Consequently the predicates that multi-valued indexes can accelerate are limited to the predicates that atomic B+ tree indexes can accelerate (i.e. equality and range predicates). The following sections will use equality and range predicates when describing an applicable query, but it is important to note that R tree indexes could also be naturally extended to handle multi-valued fields of spatial data. The separation between logical plans (the Algebricks layer) and physical plans (the Hyracks layer) is what enables such an extension.

4.3.1 Explicit Unnesting Queries

Given that we create multi-valued indexes using `UNNEST` clauses, it follows that `UNNEST` queries with an equality or range predicate on the index endpoints can utilize the index itself. Suppose we want to find all orders that have an item of quantity 100:

```
FROM     Orders O
UNNEST   O.items I
WHERE    I.qty = 100
SELECT   DISTINCT O;
```

Listing 4.23: `UNNEST` query example.

The query in Listing 4.23 involves one `UNNEST` on the `items` object array, with an equality predicate on the `qty` field. This query can be accelerated using either of the indexes given in Listing 4.24.

```
CREATE INDEX ordersItemQtyIdx ON Orders (
    UNNEST items SELECT qty
);

// Prefix will be used.
CREATE INDEX ordersItemQtyPriceIdx ON Orders (
    UNNEST items SELECT qty, selling_price
);
```

Listing 4.24: Indexes used to accelerate the queries in Listing 4.23, Listing 4.25, Listing 4.26, and Listing 4.28.

The inclusion of the second index in Listing 4.24 demonstrates that the applicability of multi-valued B+ tree indexes go beyond just queries that contain *all* fields in the index itself. Similar to our existing atomic B+ tree indexes, if a multi-valued index contains a subset of the fields in a query *and* all fields of the subset form the prefix of index itself, the index is deemed applicable for said query.

4.3.2 Membership / Existential Quantification Queries

The query in Listing 4.23 describes one approach to quantifying a value within a multi-valued field. While valid, a more natural way to express such a query is through an explicit existential quantification expression:

```
FROM    Orders O
WHERE   SOME I IN O.items SATISFIES I = 100
SELECT  O;
```

Listing 4.25: Explicit existential quantification query example.

The syntax for quantification is general enough to allow for a variety of equality and range predicates. If we have a equality predicate though, we can simplify our query further by transforming the quantification expression into a membership expression. To say that x is a member of some collection C is equivalent to saying that there exists some $c \in C$ such that $c = x$. The transformed query from existential quantification to membership is given below:

```
FROM    Orders O
SELECT  O
WHERE   100 IN O.items;
```

Listing 4.26: Membership (implicit existential quantification) query example.

Similar to Listing 4.23, both queries above can utilize any of the indexes given in Listing 4.24.

4.3.3 Universal Quantification Queries

The queries described up to this point have all dealt with *existential* quantification, though this is not the only type of quantification offered in AsterixDB. Modifying the original intent of the queries in Listing 4.23, Listing 4.25, and Listing 4.26, assume that we want instead to find all orders that have *all* items with `qty = 100`. We transform the explicit existential quantification query in Listing 4.25 to use the `EVERY` clause instead of the `SOME` clause:

```
FROM    Orders O
WHERE   EVERY I IN O.items SATISFIES I.qty = 100
SELECT  O;
```

Listing 4.27: Not applicable universal quantification query example.

As the query in Listing 4.27 stands, it is *not* applicable for acceleration with any multi-valued index. To be eligible for acceleration from the indexes in Listing 4.24, the universal quantification query must additionally specify that the multi-valued field is not empty:

```
FROM    Orders O
WHERE   EVERY I IN O.items SATISFIES I.qty = 100 AND
        LEN(O.items) > 0
SELECT  O;
```

Listing 4.28: Applicable universal quantification query example.

The non-emptiness requirement stems from the way that `NULL` and `MISSING` values are currently handled in AsterixDB with respect to atomic indexes. For a more in-depth explanation, refer to Subsection 5.5.3.

4.3.4 Join Queries With Multi-Valued Fields in the Join Predicate

The last type of query that multi-valued indexes are meant to accelerate are queries with multi-valued fields in join predicates. By having an applicable multi-valued index here, a potentially much faster join algorithm (index nested loop join) becomes available for the optimizer to choose. In the example below, we want to find all orders that contain products with the letter `"B"` in their name:

```
FROM         Products P
INNER JOIN   (
    FROM     Orders O
    UNNEST   O.items I
    SELECT   O, I.product_id
) AS OI
ON           OI.product_id /*+ indexnl */ = P.product_id
WHERE        P.name LIKE "%B%"
SELECT       DISTINCT OI.O;
```

Listing 4.29: Join query example 1.

We can also choose to rewrite Listing 4.29 with implicit `JOIN` and `UNNEST` clauses (resulting in a less cluttered query):

```
FROM     Products P,
         Orders O,
         O.items I
WHERE    I.product_id /*+ indexnl */ = P.product_id AND
         P.name LIKE "%B%"
SELECT   DISTINCT O;
```

Listing 4.30: Join query example 2.

Both join queries in Listing 4.29 and Listing 4.30 can be accelerated with an index on the product_id field inside the items object array of the Orders dataset:

```
CREATE INDEX orderItemProductIdx ON Orders (
    UNNEST items SELECT product_id
);
```

Listing 4.31: Applicable index for the queries in Listing 4.29 and Listing 4.30.

Chapter 5

Indexing Implementation

The previous chapter discussed all of the information that a user must minimally know to use multi-valued indexes in AsterixDB. This chapter aims to address all of the implementation details and is divided into six sections: (i) how to represent an index in metadata, (ii) how to physically represent an index entry, (iii) how to bulk load an index, (iv) how to maintain an index, and (v) how to recognize index applicability and utilize indexes in queries.

5.1 Metadata Representation of an Index

As with most database systems, AsterixDB stores information about its indexes inside the same unit of data collection that it manages. Such a technique enables the database itself to elegantly persist and retrieve index metadata as it would any other data collection for use at system runtime. In our case this unit is a `DATASET`, and this section aims to describe the records that detail multi-valued indexes inside the `Index` metadata dataset.

Revisiting the examples from the previous chapter, we first take a look at the metadata for the Listing 4.17 index on the `number` and `kind` fields within the `phones` object array of the

Users dataset. Once the aforementioned CREATE INDEX statement is executed, the record in Listing 5.1 is inserted into the Index dataset.

```
{
  "DataverseName": "ShopALot",
  "DatasetName": "Users",
  "IndexName": " usersNumberKindIdx",
  "IndexStructure": "ARRAY",
  "SearchKeyElements": [
    { "UnnestList": [ [ "phones" ] ],
      "ProjectList": [ [ "number" ], [ "kind" ] ] }
  ]
  "SearchKeyType": [ [ "string", "string" ] ]
}
```

Listing 5.1: Metadata for a composite multi-valued index with type definitions.

There are three fields of interest here:

1. IndexStructure indicates the type of index that this record describes. The value of "ARRAY" informs us that this is a multi-valued index.
2. SearchKeyElements is an array of objects describing the UNNEST and SELECT portions of the index specification. UnnestList is a two-level array, with the outer level containing paths to multi-valued fields and the inner level describing the fields of the aforementioned paths. ProjectList is another two-level array, the outer describing paths to the atomic endpoints and the inner describing fields of the aforementioned paths.
3. SearchKeyType is a two-level array describing the types of each index element. This field only exists if type definitions exist in the index specification. The outer level maps items in the SearchKeyElements field to a collection of type specifications via their position in the array. It is essential to keep the index element abstraction present to fully support composite multi-valued indexes with atomic elements in the future (given the specifications in Listing 4.19 and Listing 4.20). The inner level of SearchKeyType

maps items in a ProjectList to type specifications via their position in the array. If there is no ProjectList (i.e. no SELECT present in the specification), then the sole item in an inner array maps to the last item in the UnnestList.

To motivate SearchKeyElements being an array of objects (as opposed to a sole object), we revisit the composite atomic and multi-valued field index specification in Listing 4.19. Listing 5.2 describes the Index metadata dataset record for such an index.

```
{
  "DataverseName": "ShopALot",
  "DatasetName": "Stores",
  "IndexName": "storesZipCatIdx",
  "IndexStructure": "ARRAY",
  "SearchKeyElements": [
    { "ProjectList": [ [ "address", "zip_code" ] ] },
    { "UnnestList": [ [ "categories" ] ] }
  ]
}
```

Listing 5.2: Metadata for a composite index containing an atomic element and a multi-valued element.

For such a composite index, there are now *two* objects in the SearchKeyElements array. The first object describes the atomic field zip_code inside the address object, containing no UNNEST clauses. The second object describes the multi-valued element, containing the multi-valued field categories. Note the absence of the ProjectList field in the second object, which corresponds to the absence of a SELECT clause in the index specification.

To motivate the two-level aspect of the UnnestList and ProjectList arrays, suppose that the type definition DDL for Orders was modified to include an info field for each object in the items array. The Orders type definition DDL in this scenario is given in Listing 5.3. To create a composite index on the info and item.qty fields inside the items array, one would use the index specification in Listing 5.4, yielding the Index metadata dataset record in Listing 5.5.

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

Listing 5.3: Type definition for a dataset for composite fields within multi-valued fields.

```
CREATE INDEX ordersLineInfoQtyIdx ON Orders (
    UNNEST items SELECT info, item.qty
);
```

Listing 5.4: Example index specification for a multi-valued element with an atomic endpoint located within an object.

```
{
  "DataverseName": "ShopALot",
  "DatasetName": "Orders",
  "IndexName": "ordersLineInfoQtyIdx",
  "IndexStructure": "ARRAY",
  "SearchKeyElements": [
    { "UnnestList": [ [ "items" ] ],
      "ProjectList": [ [ "info" ], [ "item", "qty" ] ] }
  ]
}
```

Listing 5.5: Metadata for a composite index containing a multi-valued element with an atomic endpoint located within an object.

Note how the structure of `ProjectList` makes it clear what our atomic endpoints are: (i) the `info` field and (ii) the `qty` field inside the `item` object.

To create an index on the multi-valued field on the item.tags array inside the items object array, the index specification in Listing 5.6 would yield the Index metadata dataset record in Listing 5.7.

```
CREATE INDEX orderItemTagIdx ON Orders (
    UNNEST items UNNEST item.tags
);
```

Listing 5.6: Example index specification with a multi-valued field within an object.

```
{
  "DataverseName": "ShopALot",
  "DatasetName": "Orders",
  "IndexName": "orderItemTagIdx",
  "IndexStructure": "ARRAY",
  "SearchKeyElements": [
    { "UnnestList": [ [ "items" ], [ "item", "tags" ] ] }
  ]
}
```

Listing 5.7: Metadata for an index with a multi-valued field within an object.

Similarly, note how the structure of UnnestList makes it clear what our multi-valued fields are: (i) the items array and (ii) the tags array inside the item object.

5.2 Representation of an Index Entry

Having just described the metadata representation of an index itself, we now describe how an index *entry* is actually represented. This section will describe index entries for a multi-valued index of type BTREE, but again it is important to stress that this work is general enough to also be applied to RTREE indexes.

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 payload (data record(s) or

```
{
  "store_id": "C1",
  "name": "Velda's Pantry",
  "categories": [ "Produce", "Breakfast" ]
},
{
  "store_id": "C2",
  "name": "Sheetz Bakery",
  "categories": [ "Bread & Bakery", "Deli" ]
},
{
  "store_id": "C3",
  "name": "Sheetz Bakery",
  "categories": [ "Bread & Bakery", "Bread & Bakery", null ]
},
{
  "store_id": "C4",
  "categories": [ ]
}
```

Listing 5.8: Sample documents for the `Stores` dataset.

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 the B+ tree in the presence of potentially duplicate secondary B+ tree key values is maintained by adding the record's primary key as a suffix to the sort key itself. To illustrate these points, suppose that the documents in Listing 5.8 are to be inserted into the `Stores` dataset with an atomic B+ tree index on the `name` field. The corresponding `name` index would contain the following entries in the given order:

1. `("Sheetz Bakery", "C2")`
2. `("Sheetz Bakery", "C3")`
3. `("Velda's Pantry", "C1")`

In terms of string comparison, `"Sheetz Bakery"` is less than `"Velda's Pantry"`, hence the `"C1"` entry comes after the other two entries. To handle duplicate key entries, we compare the primary key fields among the duplicates (`"C2"` being less than `"C3"`) and put `"C2"` before

"C3". AsterixDB chooses not to store NULL or MISSING values in any of its indexes, therefore there exists no entry for record "C4".

To make multi-valued indexes compatible with index types other than B+ trees, an index entry in a multi-valued B+ tree index is really no different than an index entry in an atomic B+ tree index. Multi-valued indexes instead work above the storage component in AsterixDB (the Job Execution and SQL++ Compiler components from Figure 3.1). In the multi-valued field of an index, the sort key is drawn from the *atomic endpoints* (as opposed to the enclosing multi-valued field itself). Using the index on the categories string array of the Stores dataset as an example, the atomic endpoints are the individual items of the categories array (e.g. "Produce", "Breakfast", etc...) and the categories array itself is the enclosing multi-valued field.

The approach of using atomic endpoint(s) introduces two new issues: (i) how to handle duplicate values within a multi-valued field, and (ii) how to handle empty multi-valued fields. Starting with the issue in item (i), recall that a primary key appears at most once in an atomic index. Such a statement is no longer true with multi-valued indexes, as a primary key value can now be associated with multiple index entries. This non-uniqueness leads to several issues when presented with duplicate values in the multi-valued field, the most notable being concurrency (discussed in Subsection 5.4.1). Given that multi-valued covering indexes are not in the scope of this thesis (due to the implementation challenges described in Appendix B), the question arises: "Is it even necessary to store duplicates?". Every single query in Section 4.3 can be answered without the inclusion of duplicate values, so the decision was made to simply not store duplicates in the first place to solve the issue in item (i). The issue in item (ii) has two solutions: either store the index entry with a special empty value for a multi-valued field or do not store the index entry at all. The former would allow universal quantification queries without a non-emptiness clause to be accelerated by a multi-valued index (see Subsection 5.5.3 for why), but it would require storage layer changes to accommodate the aforementioned special value. Such an issue draws parallels to how the values `MISSING` and `NULL` are treated with respect to index entries. AsterixDB chooses not to store such values in any index, so the decision was made to not include records with empty multisets or arrays in multi-valued indexes.

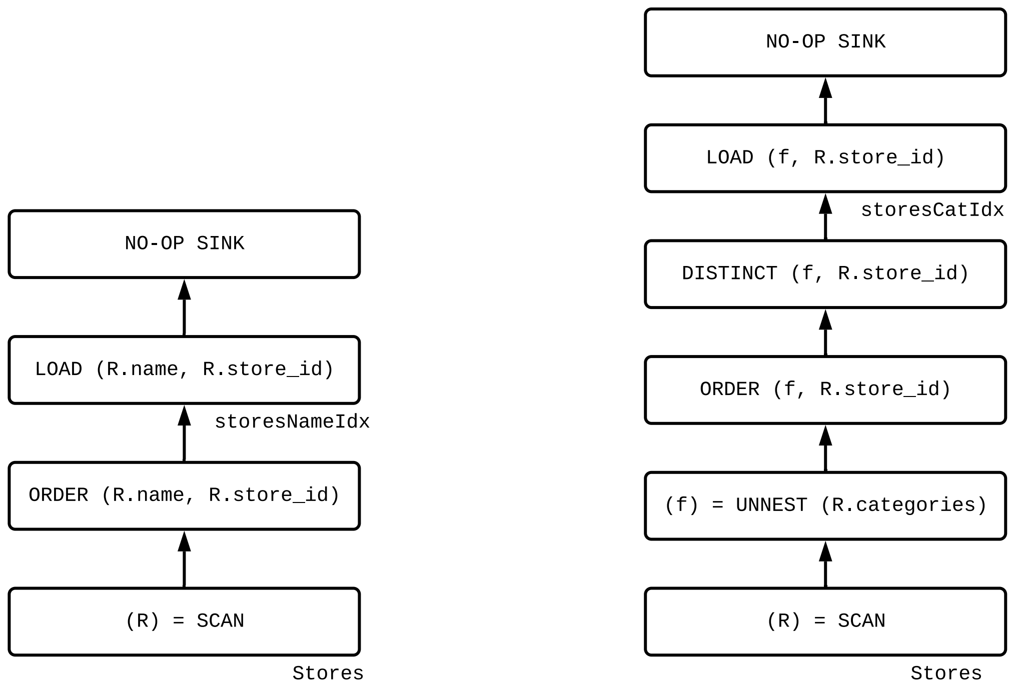
Using the documents in Listing 5.8 and an index on the `categories` string array field for the `Stores` dataset (specification in Listing 4.18), the corresponding index would contain the following entries in the given order:

1. `("Bread & Bakery", "C2")`
2. `("Bread & Bakery", "C3")`
3. `("Breakfast", "C1")`
4. `("Deli", "C2")`
5. `("Produce", "C1")`

All entries are first sorted by the individual items in `"categories"`, and then by their primary key field `"store_id"`. To handle duplicates among the `"categories"` array items, the primary key is used to impose a total order. There exists two items for `"Bread & Bakery"` in the `"C3"` record, but only one is recorded in the index itself. Similar to atomic indexes, the `NULL` value in the `"C3"` record's `categories` array is not recorded in the index. Finally, the `"C4"` entry is not stored in the index as the `categories` array itself is empty.

5.3 Bulk Loading an Index

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. All other cases fall under index maintenance (Section 5.4). This section aims to describe the data flow to accomplish the task of bulk-loading a multi-valued index.



(a) Example data flow for an atomic index. (b) Example data flow for a multi-valued index.

Figure 5.1: Example Hyracks jobs for bulk-loading indexes.

5.3.1 Create Index Statements

When a `CREATE INDEX` statement is issued from a SQL++ client, there are three main steps taken: (i) The `CREATE INDEX` statement is parsed and the `Index` dataset metadata record is constructed. (ii) The metadata record is inserted into the `Index` dataset. (iii) The existing data from the dataset that the index is being created on is bulk-loaded into the index based on the metadata from step (i). Using the existing operators provided by the Hyracks runtime (through the Job Execution component of Figure 3.1), the goal of bulk-loading is to assemble a DAG of operators that passes sorted index entries to a bulk-load operator that will handle the actual B+ tree construction in parallel across each AsterixDB cluster node. Figure 5.1 describes two such DAGs for indexes on the `Stores` dataset. Figure 5.1a details the data flow for bulk-loading the atomic index on the `name` field. We first `SCAN` the `Stores` dataset for all records R. The output of this operator is then given to the `ORDER` operator above, sorting all (R.`name`, R.`store_id`) pairs. Finally, the output of the `ORDER` operator is given to the LOAD operator where the B+ tree is constructed. All DAGs must end with an explicit SINK operator node, which will consume the (R.name, R.store_id) pairs from the LOAD operator.

Figure 5.1b describes a DAG for bulk-loading a multi-valued index on the categories field of the Stores dataset. In contrast to Figure 5.1a, we now include an UNNEST on the multi-valued field before the ORDER. This UNNEST produces the individual items of R.categories, which are assigned the variable f. If the index specification called for more UNNEST clauses, then additional UNNEST operators would be applied above. Now having the sort key f and the primary key R.store_id, all (f, R.store_id) pairs are sorted by the ORDER operator. A DISTINCT operator is applied immediately after the ORDER to remove duplicate entries within the multi-valued fields. These pairs are then given to the LOAD operator and finally consumed by the SINK operator.

5.3.2 Load Statements

A LOAD statement reads a file of records and inserts all of its records into the corresponding dataset. In contrast to handling a single CREATE INDEX statement, one must be able to accommodate *all* indexes on a dataset when a LOAD statement is issued. A second notable difference is that the LOAD statement in AsterixDB operates at the Algebricks layer as opposed to the Hyracks layer, giving us access to a different set of operators. All future data flows in this chapter work at the Algebricks layer, as this allows any plan detailed here to be created and manipulated by the query optimizer.

Figure 5.2 describes the LOAD data flow for the Stores dataset with the two indexes from the previous subsection (storesNameIdx and storesCatIdx). Records R are first read from the load file and then passed to the next operator in the pipeline. All records are then sorted by their primary key R.store_id before being given to the LOAD operator

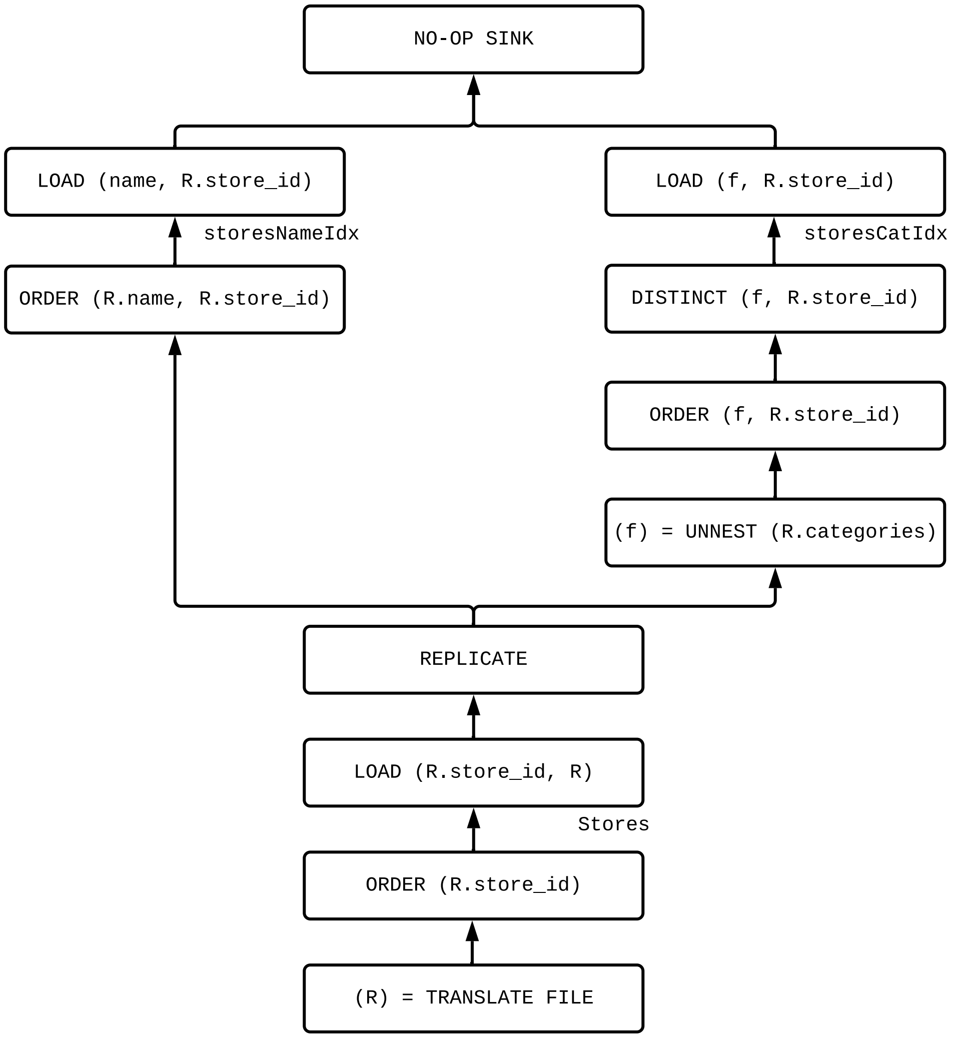


Figure 5.2: Example data flow (Algebricks job) for a bulk-load.

to populate the Stores dataset itself. To perform the bulk-loading of secondary indexes in parallel, a REPLICATE operator is used after extracting the multi-valued starting points (sort keys). The data flow for each branch after the REPLICATE is equivalent to the data flow shown after the dataset SCAN operator in the previous subsection. The input to the storesNameIdx REPLICATE branch is the (R.name, R.store_id) pair, which is ordered and consequently given to its appropriate LOAD operator. The input to the storesCatIdx branch is the (R.categories, R.store_id) pair, which goes through the UNNEST, ORDER, and DISTINCT operators before reaching the LOAD operator for the storesCatIdx index. Outputs from both LOAD operators are then consumed by the SINK. To accommodate more secondary indexes (atomic or multi-valued) the data flow would simply have another parallel branch between the `REPLICATE` and `SINK` operators.

5.4 Maintaining an Index

`INSERT`, `DELETE`, and `UPSERT` (insert if the document does not exist, replace the document if it does) are the three maintenance operations in AsterixDB. This section describes the constraints and data flows associated with these operations.

5.4.1 Locking in AsterixDB

Before discussing the data flow associated with maintenance operations, we must first discuss the transaction operational flow our that maintenance operations are constrained by. A database transaction is a sequence of operations that is abstracted to a singular logical unit of work. Transactions have four properties associated with them, described in the acronym *ACID*:

1. *Atomicity*: All or no operations are performed.
2. *Consistency*: No integrity constraints are violated after completion.
3. *Isolation*: A concurrent execution of multiple transactions is equivalent to the result of some serial sequence of the same transactions.
4. *Durability*: Effects of the operations are persisted after the transaction completes.

In AsterixDB transactions are (i) of record-level granularity, (ii) local to each cluster node, and (iii) act across a dataset's primary and secondary indexes. Suppose the following `INSERT` statement were issued on the `Users` dataset in an AsterixDB instance with two cluster nodes N_1 and N_2:

```
INSERT INTO Users [
    { "user_id": "C1", "name": "John" },
    { "user_id": "C2", "name": "Mary" }
];
```

Listing 5.9: Example `INSERT` statement of two documents.

Regardless of which cluster nodes the records are directed to, there will always be two transactions here: one for the `"C1"` record and another for the `"C2"` record. If the `"C1"` record is directed to node N_1 and the `"C2"` record is directed to node N_2, one local transaction would occur at N_1 and another local transaction would occur at N_2. If the `"C1"` `INSERT` were to fail but the `"C2"` `INSERT` were to succeed, then no action would be taken by AsterixDB to remove the `"C2"` record. Handling such a situation differently would require non-local (i.e. distributed) transactions. If this scenario occurs, the user is left to handle the after-effects.

To realize the *Isolation* property of transactions, locks must be acquired for the resources that the operations of a transaction requires. Note that locking is introduced in this section and not the previous because bulk-loaded records cannot be accessed by other operations until the bulk loading itself is finished, thus isolation is implicitly achieved. Operations in a transaction can be broadly divided into two groups: (i) operations that read and (ii) operations that write. Exactly what type of transactions can acquire locks and the duration locks can be held for are typically varied in different systems to achieve better performance for different types of workloads. In AsterixDB, 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 such 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 for accessing secondary indexes. No locking here means that a read operation on a secondary index is allowed to 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), 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.

5.4.2 Insert Statements

The goal of an `INSERT` statement is to insert the given record(s) into the appropriate primary index and all associated secondary indexes. Recall that to realize bulk-loading for multi-valued indexes, a sequence of `UNNEST` operators were required to extract the atomic endpoint values. Thus to execute all index maintenance operations, `UNNEST` operators must be added to the existing data flow for atomic indexes.

Figure 5.3 describes the data flow for an `INSERT` statement on the `Orders` dataset with three secondary indexes: (i) an atomic index on the `user_id` field, (ii) a multi-valued index on the `tags` items inside the `items` object array, and (iii) a multi-valued index on the `qty` field inside the `items` object array. Starting from the bottom source operator, the primary key R.`order_id` is extracted from the input record(s) and is given to the primary index `INSERT` operator along with the record itself. Primary index maintenance operations are always performed before secondary index maintenance operations to obtain locks to prevent inconsistencies that could stem from other transactions on the secondary indexes themselves. Once the corresponding record is inserted into the primary index, we extract the fields required for our secondary index insertions. First an `INSERT` into the atomic `orderUserIDIDx` index, next an `INSERT` into the multi-valued `orderItemTagIdx`, followed by an `INSERT` into the final multi-valued `orderItemsQtyIdx`, and finally the primary key `order_id` is given the `COMMIT` operator to release the lock on the primary index.

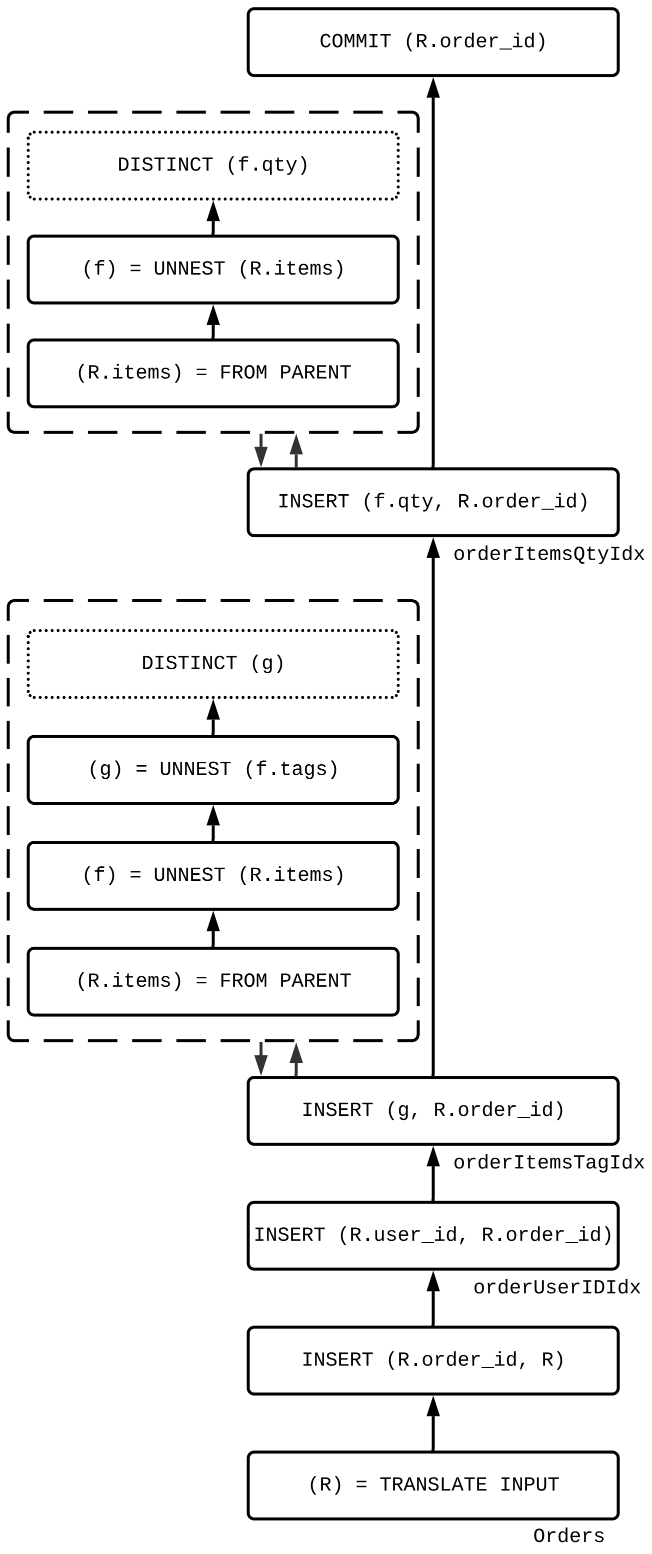


Figure 5.3: Example data flow for an `INSERT` statement.

Attached to each multi-valued `INSERT` operator is a *subplan*, a separate DAG of operators that accepts the input from its parent operator and returns some output to that same parent. The results of this separate DAG are local to its enclosing `INSERT` operator. The same data flow to extract the atomic endpoints is then applied with each subplan. The dotted lines around the `DISTINCT` operator after the top-level `UNNEST` operator in each subplan indicate that this operation is performed by the *Hyracks Dataflow* system component of each cluster node. This `DISTINCT` will remove duplicate secondary key values before handing the values off to the parent `INSERT`. Once all secondary key values are received by the parent, the input record is forwarded to the next operator in the pipeline. The reason why multi-valued `INSERT` DAGs cannot exist outside of a subplan is because of the output cardinality change associated with `UNNEST` operator(s). To motivate the use of subplans here, let us briefly explore an alternative means of performing an `INSERT` with two multi-valued indexes. This alternative must satisfy the following properties:

1. The lock on the primary index must not be released until all secondary index entries are inserted for that transaction. In terms of the data flow, this means that primary keys cannot be allowed to reach the `COMMIT` operator prematurely.
2. All primary key values given to the transaction `COMMIT` operator must be unique.
3. The number of input records into the subgraph for handling the multi-valued index operation must be equal to the number of output records out of said subgraph.

The alternative approach considered here is to use a single path to handle the entire `INSERT` data flow. An example of the relevant portions of the alternative data flow is given in Figure 5.4. To satisfy all of the requirements above, an `ORDER` operator followed by a `DISTINCT` operator is applied on the dataset's primary key after each multi-valued index's `INSERT`. The inclusion of the `ORDER` operator satisfies the first requirement because `ORDER` is *blocking*. This means that no records will be given to the next operator before all records below are finished. For the final multi-valued index `INSERT` in the entire data flow, this means all

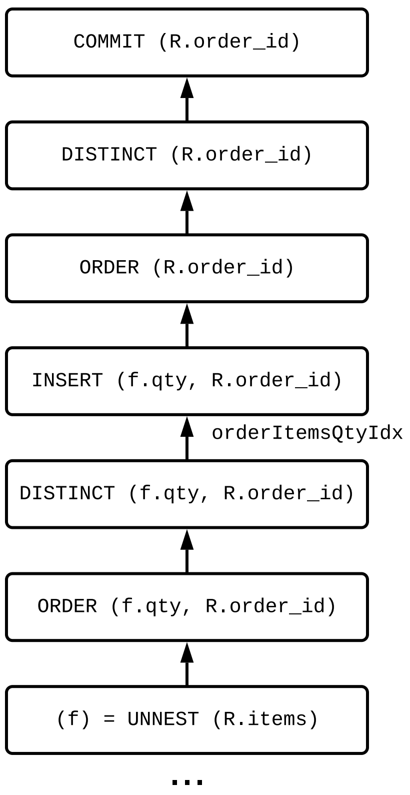


Figure 5.4: Alternative partial data flow for an INSERT statement.

records must be given to the ORDER before any primary key reaches COMMIT. The DISTINCT operator satisfies the second and third requirements as the cardinality increases from the previous UNNEST operator(s) are reverted.

The big disadvantage of this potential alternative approach is the inclusion of the blocking ORDER operator. For INSERTs with a large number of records, the locks on the primary index entries for these records are held until *all* records are finished being processed. This contrasts with the subplan approach, where after a single input record is processed, said record can be handed off to the next operator (such as another index INSERT operator or the COMMIT). Though the alternative ORDER + DISTINCT approach was originally used, the absence of a blocking operator in the subplan approach is the main reason for the data flow now used and presented in Figure 5.3.

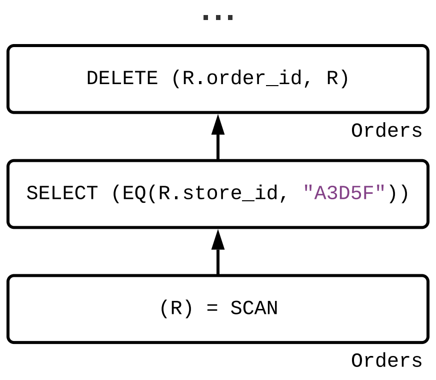


Figure 5.5: Data flow for the search phase of a DELETE statement.

5.4.3 Delete Statements

A DELETE operation has two phases: (i) a search phase (to find all qualifying records), and (ii) a delete phase. The storage-level API call to delete an entry is nearly identical to the API call to insert an entry. Hence the data flow for a DELETE operation is *identical* to the data flow for an INSERT operation except for the inclusion of this search phase. This is illustrated in the Figure 5.5, describing the data flow for the following DELETE statement:

```
DELETE FROM      Orders
WHERE            store_id = "A3D5F";
```

Listing 5.10: DELETE statement associated with Figure 5.5.

In Figure 5.5, only one new operator has been added for the search phase: a SELECT operator to filter out all records that do not qualify for the delete itself. The rest of the data flow remains the same as the INSERT data flow of Figure 5.3, replacing all INSERT operators with DELETE operators.

5.4.4 Upsert Statements

An UPSERT on a dataset with *atomic* secondary indexes can be described in three steps:

1. Given some incoming record R_{new} with primary key k, search the primary index for an existing record R_{old} using k.
2. Perform the UPSERT for the primary index. If R_{old} does not exist, then a storage-level insert operation is performed for the incoming record R_{new} into the primary index. If R_{old} does exist, then a storage-level delete operation is performed to remove the existing record R_{old} before performing an insert for the incoming record R_{new} into the primary index.
3. Perform the UPSERT for the secondary indexes. If R_{old} does not exist, then again a storage-level insert operation is performed into the secondary indexes (if the indexed field exists in the incoming record R_{new}). If R_{old} does exist, then the actions to be done are split depending on the contents of the existing record R_{old} and the incoming record R_{new}:
   (a) If the indexed field exists in the existing record R_{old} but not in the incoming record R_{new}, then a delete operation is issued to the secondary index.
   (b) If the indexed field exists in the incoming record R_{new} but not in the existing record R_{old}, then an insert operation is issued to the secondary index.
   (c) If the indexed field exists in both records R_{old} and R_{new} but differs in value, then a delete operation for the existing record R_{old} followed by an insert operation for the incoming record R_{new} is issued to the secondary index.
   (d) If the indexed field exists in both records R_{old} and R_{new} but are both equal in value, then no operation is performed.

In terms of the data flow required for UPSERT, there are now two record variables to deal with: the existing record R_{old} and the record given in the UPSERT statement itself R_{new}. Integrating these for multi-valued indexes using the aforementioned subplan approach can be found in the Figure 5.6 example. Here, the primary index maintenance operator (the UPSERT operator for Orders) now has an output: the existing record R_{old}. The search for this old record R_{old} such that the order_id in R_{old} is equal to the order_id in the new record

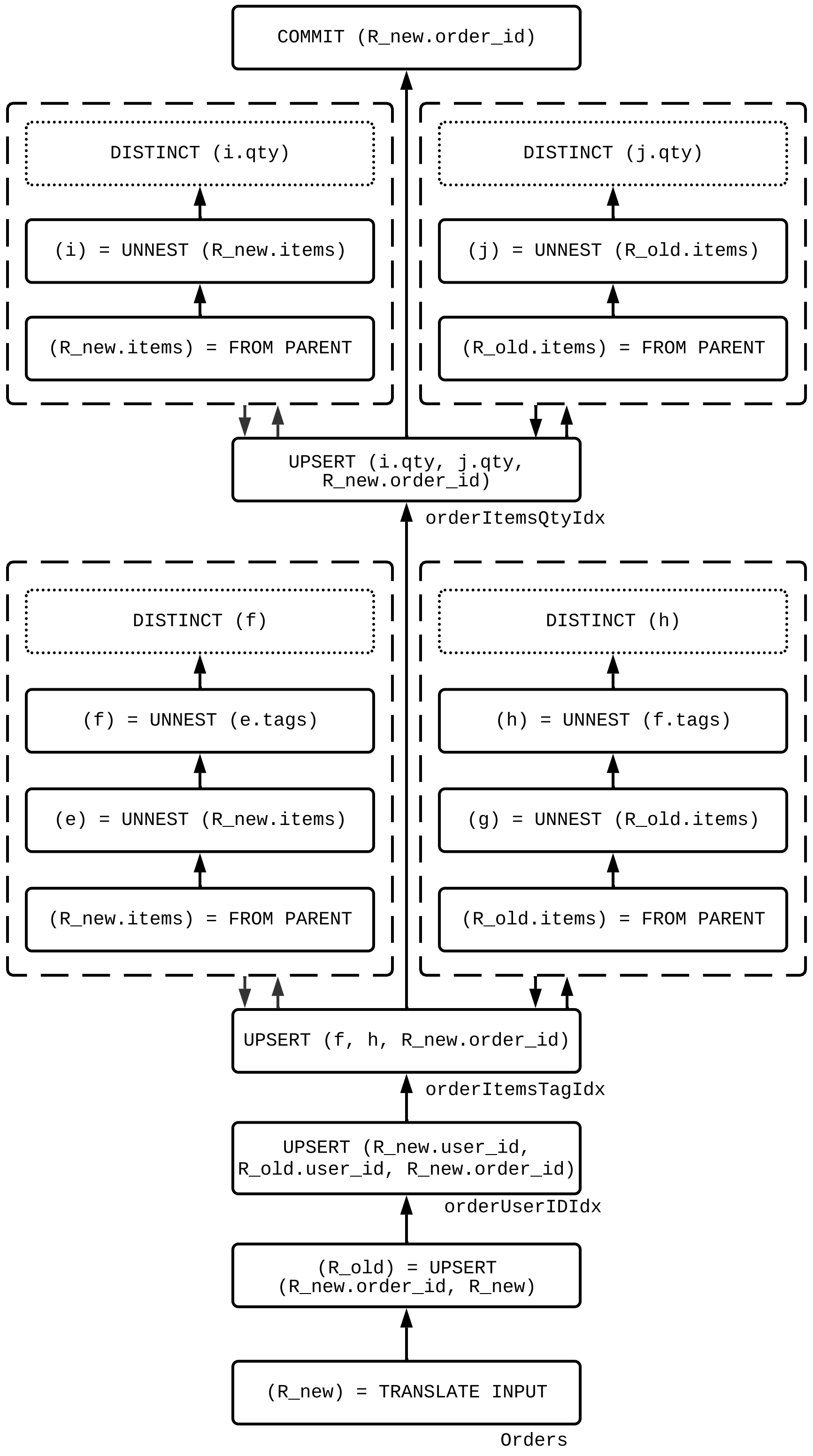

COMMIT (R_new.order_id)
DISTINCT (i.qty)
(i) = UNNEST (R_new.items)
(R_new.items) = FROM PARENT
DISTINCT (j.qty)
(j) = UNNEST (R_old.items)
(R_old.items) = FROM PARENT
UPSERT (i.qty, j.qty, R_new.order_id)
orderItemsQtyIdx
DISTINCT (f)
(f) = UNNEST (e.tags)
(e) = UNNEST (R_new.items)
(R_new.items) = FROM PARENT
DISTINCT (h)
(h) = UNNEST (f.tags)
(g) = UNNEST (R_old.items)
(R_old.items) = FROM PARENT
UPSERT (f, h, R_new.order_id)
orderItemsTagIdx
UPSERT (R_new.user_id, R_old.user_id, R_new.order_id)
orderUserIDIdx
(R_old) = UPSERT (R_new.order_id, R_new)
(R_new) = TRANSLATE INPUT
Orders

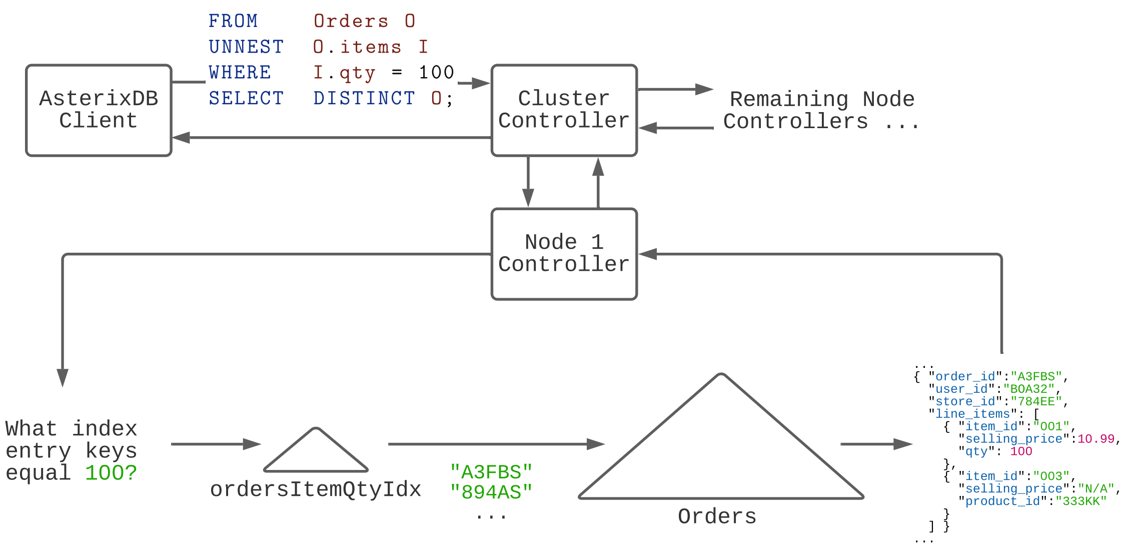
Figure 5.6: Example data flow for an UPSERT statement.

R_{new} (i.e. the primary key values are equal) is not performed at the Algebricks layer, rather this search is performed in the Hyracks Dataflow system component (within the primary index `UPSERT` operator itself). The index fields `user_id` from R_{old} and `user_id` from R_{new} are given to the `UPSERT` operator for the atomic index `orderUserIDIdx`. The `UPSERT` logic in Item 3 above is then performed.

We now move to the next operator in the sequence. Both records R_{old} and R_{new} are given to the `UPSERT` operator for the multi-valued index `orderItemsTagIdx`, which passes R_{new} to the left subplan to extract the new `tags` array values, f, and R_{old} to the right subplan to extract the old `tags` array values, h. In contrast to the `UPSERT` logic applied in the atomic index `UPSERT` operator itself, the multi-valued `UPSERT` operator performs an unconditional delete of all the old index entries associated with the working R_{old} record followed by an insert of all the new index entries associated with R_{new}. The same multi-valued `UPSERT` logic is applied for the next index `ordersItemQtyIdx` as well, extracting the sort key values from R_{old} and R_{new} using two subplans and performing the same unconditional delete and insert. Once all secondary index `UPSERT` operations are finished, the primary key `order_id` is given the `COMMIT` to finish the transaction for that specific record. Potential future work with respect to handling `UPSERT`s could involve introducing a filter within the multi-valued `UPSERT` operator itself to avoid needless work when multi-valued field values are unchanged.

5.5 Optimizing Queries

Given any Algebricks-level data flow (henceforth referred to as a query plan), the goal of the query optimizer is to transform the query plan given 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. This section discusses the how the applicability of a multi-valued index


```
FROM     Orders O
UNNEST   O.items I
WHERE    I.qty = 100
SELECT   DISTINCT O;
```

```
...
{ "order_id":"A3FBS",
  "user_id":"BOA32",
  "store_id":"784EE",
  "line_items": [
    { "item_id":"001",
      "selling_price":10.99,
      "qty": 100
    },
    { "item_id":"003",
      "selling_price":"N/A",
      "product_id":"333KK"
    }
  ] }
...
```

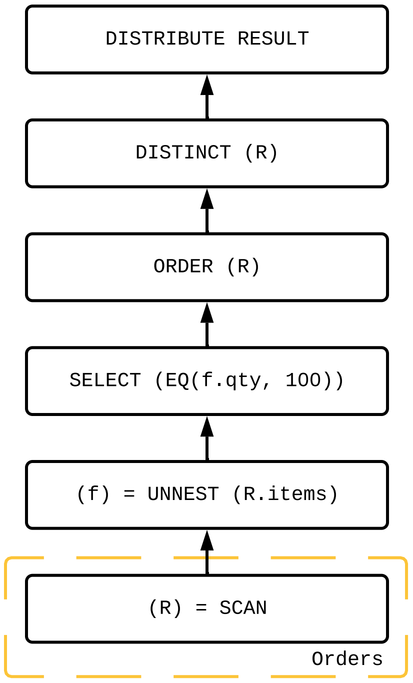
Figure 5.7: High-level view of an accelerated UNNEST query plan.

is recognized for some query plans and how said query plan is transformed to utilize the applicable multi-valued index.

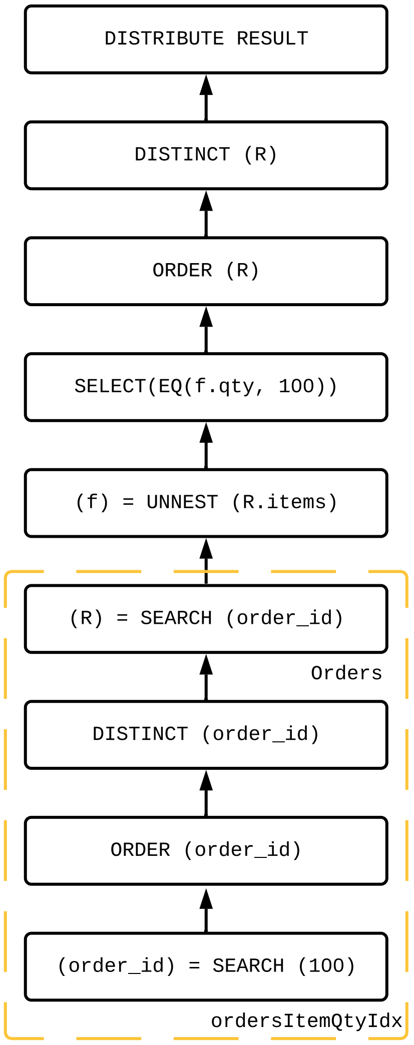
5.5.1 Explicit Unnesting Queries

We start our discussion about detailed query optimization with the explicit UNNEST query from the previous chapter (Listing 4.23). Figure 5.7 details how we can utilize a secondary index to accelerate this query at a high level. The job executed at each AsterixDB cluster node entails (i) searching the secondary index for entries whose sort key satisfies the `I.qty = 100` predicate, (ii) using the primary key of these entries to search the primary index for records, and (iii) giving these records to the remainder of the plan.

Delving into the details now, without such a multi-valued index, the plan on the left of Figure 5.8 would be chosen. A full data scan of `Orders` is performed, followed by an `UNNEST` operator to extract the relevant fields, with records filtered using the `SELECT` operator, and the results being passed through an `ORDER` operator and `DISTINCT` operator to satisfy the



(a) Example explicit `UNNEST` query plan.



(b) Example accelerated `UNNEST` query plan.

Figure 5.8: Example query plans to demonstrate `UNNEST` query acceleration.

`DISTINCT` clause in the query before giving the result back to the user. We note that the explicit `DISTINCT` operator in AsterixDB currently requires a sorted input, so the `ORDER` operator is necessary for this specific query.

The query plan in Figure 5.8a must scan through every single record in the `Orders` dataset, perform an `UNNEST` on every single record, and evaluate the equality predicate $f.\texttt{qty} = 100$ for every item in the `items` object array. In contrast, the plan on the right (Figure 5.8b) *only* scans entries from the `ordersItemQtyIdx` index whose sort key `qty` is equal to 100. The result of this scan is the primary key associated with each index entry (`order_id`).

Because a primary key may map to more than one index entry, we remove all duplicates before the `Orders` primary index search to avoid performing unnecessary work. Once the `Orders` primary index search is performed, the secondary index validation step is executed with the `UNNEST` and `SELECT` operator. If a qualifying record changes from the time of the initial B+ tree search on the `ordersItemQtyIdx` index to this `SELECT` operator, the record will not be passed to the next operators in the pipeline. To satisfy the `DISTINCT` clause in the query, all validated records are passed through `ORDER` and `DISTINCT` operators before being sent back to the user. Examining the two query plans in Figure 5.8, the following observation can be made: the DAG after the primary index `SCAN` operator of Figure 5.8a is *identical* to the DAG after primary index `SEARCH` operator in Figure 5.8b. We can now formulate a more specific goal for index acceleration: to replace the primary index scan with a secondary index search followed by a primary index search.

Having detailed the plan differences between a full dataset scan query and a query that utilizes a secondary index, the next question we can ask is: "How do we know when to transform a primary index scan with a secondary index search + a primary index search?" With atomic indexes, applicability is recognized by examining each conjunct in every `SELECT` operator's predicate. For atomic B+ tree indexes in particular, the following criteria must be met:

1. Given some predicate in a `SELECT` operator for a query plan, there must exist some conjunct C in the aforementioned predicate.
2. C must be a function in the set: { `EQ`, `LT`, `LE`, `GT`, `GE` }
3. One argument to the function C must be a path of fields from the source dataset that can be matched to a path of fields and dataset in some `Index` metadata record.

For multi-valued B+ tree indexes, one more criterion is added: the sequence of `UNNEST` clauses and associated paths in the index specification must match the sequence of `UNNEST` operators in the to-be-optimized query plan. The argument to the function C in a `SELECT` operator's

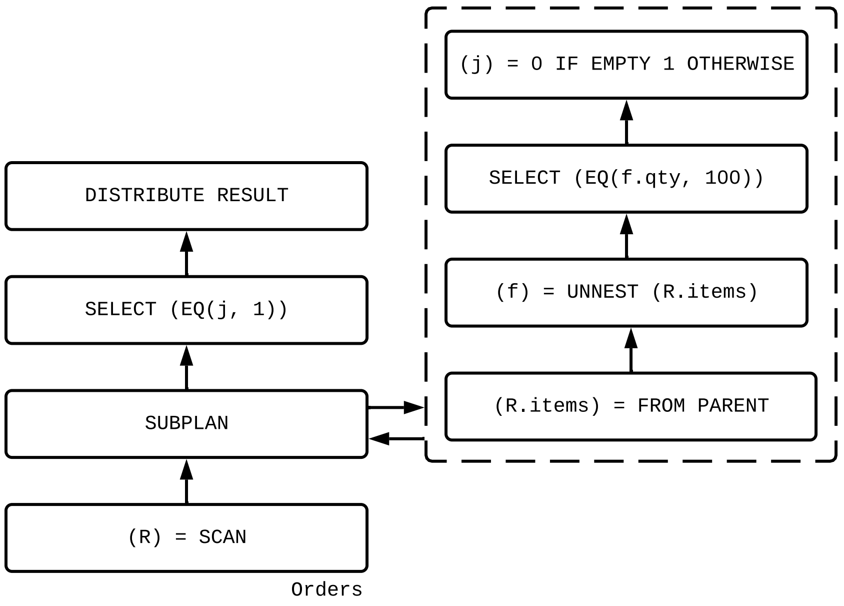


Figure 5.9: Example existential quantification query plan.

predicate must then be a path specified in the SELECT clause of the index specification or the output of the UNNEST operator if no SELECT clause exists. In Figure 5.8a, we recognize that the UNNEST operator on the items object array matches an index specification with an UNNEST on the same items array, and that the atomic endpoint of the aforementioned index specification qty can be found in an applicable conjunct of the SELECT above it.

5.5.2 Membership / Existential Quantification Queries

Figure 5.9 describes a non-indexed query plan for the existential quantification queries in the previous chapter (Listing 4.25 and Listing 4.26). After scanning the Orders dataset, the items object array is given to the SUBPLAN operator. From here, the items array goes through an UNNEST operator and a SELECT operator to satisfy the query predicate. If there exist any records after the SELECT operator, a value of 1 is returned to the SUBPLAN operator itself. If the SELECT operator filters out all records for the working items array instance, then a value of 0 is returned instead. We now move away from the subplan DAG itself. At this point (before the second SELECT operator but prior to the result distribution), a value of $j = 1$ or $j = 0$ is attached to each record R indicating whether that record has satisfied the existential predicate or not. This boolean value and the record are given to the second and final SELECT operator to return to the user qualifying records R where $j = 1$.

To reiterate, the goal of index acceleration is to replace the primary index scan (in this case, the SCAN operator on Orders) with a secondary index search followed by a primary index search. Unlike the previous subsection, the UNNEST clause and SELECT clause that we need to recognize index applicability is located within a subplan. Thus, to recognize index applicability for existential quantification queries, the following criteria is required: (i) if there exists a SUBPLAN operator, check the subplan DAG itself for all the criteria mentioned in the previous subsection, (ii) the last operator of the subplan DAG must return a single value $j = 1$ if the previous SELECT does not filter out all of the source records and $j = 0$ otherwise, and (iii) there must exist a SELECT after the SUBPLAN operator to filter out records where $j = 0$ (to prove that this subplan is used for quantification). If each criterion is met, then the same query plan transformation shown at the bottom(s) of Figure 5.8 is performed.

5.5.3 Universal Quantification Queries

Figure 5.10 describes a query plan for the universal quantification query in the previous chapter (Listing 4.28). After scanning the Orders dataset, the items object array is again given to the SUBPLAN operator. Similar to the existential quantification query plan, the items array goes through an UNNEST operator to extract the qty field. To evaluate universal quantification for the predicate EQ$(f, 100)$, the following logical translation is made:

$$\{\ \forall f \in r.\texttt{items} \mid f.\texttt{qty} = 100\ \} \ \equiv\ \{\ \nexists f \in r.\texttt{items} \mid f.\texttt{qty} \neq 100\ \} \tag{5.1}$$

The right quantification is performed with the last two operators in the subplan. The predicate inside the quantification is negated for the SELECT operator (EQ $\rightarrow$ NE) and the

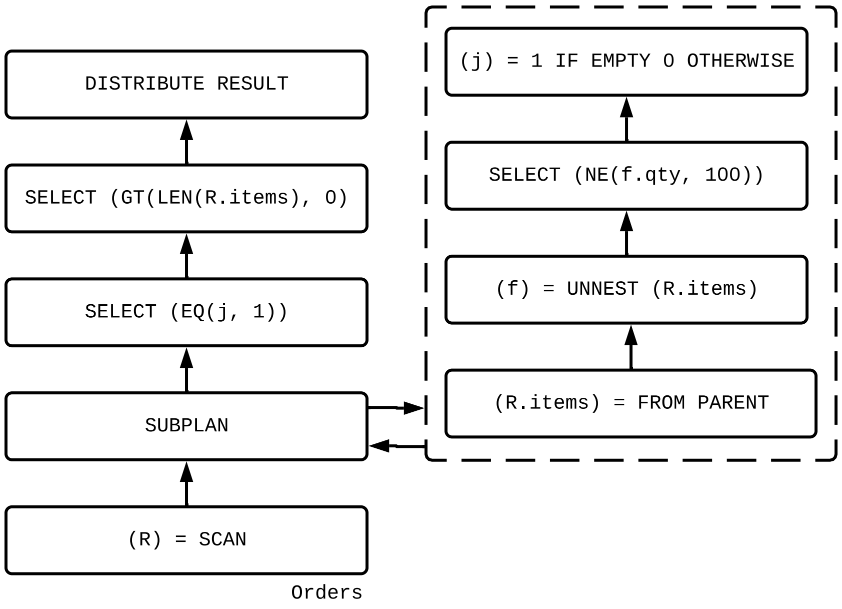


Figure 5.10: Example universal quantification query plan.

logic to assign the boolean variable j now becomes $j = 1$ if the previous SELECT removes all records and $j = 0$ otherwise. j then is given to the SUBPLAN operator, which is used to filter out records R where $j = 0$. The non-emptiness predicate on the items object array is applied on the last SELECT operator before giving the results back to the user.

To recognize applicability for universal quantification queries, the following criteria are required:

1. For all SUBPLAN subplans, check the subplan DAG itself for all criteria mentioned in Subsection 5.5.1.
2. When checking a SUBPLAN operator's subplan, additionally check for the inverse of the applicable B+ tree index functions.
3. The last operator of the subplan DAG should return $j = 0$ if SELECT does not filter out all of the source records, and $j = 1$ otherwise.
4. There must exist a SELECT after the SUBPLAN operator to filter out records where $j = 0$ (to prove that this subplan is used for quantification).
5. There must exist a non-emptiness conjunct on the multi-valued field being quantified.

If the criteria above is met, the *exact same* query plan transformation from the previous subsection is performed.

We can easily prove that such a query plan transformation is valid, starting with a universal quantification on a multi-valued field F where $|F| > 0$:

$$U = \{\ \forall f \in F \mid P(f)\ \} \tag{5.2}$$

Given the predicate P in the universal quantification above, the multi-valued index B+ tree search returns the primary keys of all records that would satisfy the existential quantification:

$$E = \{\ \exists f \in F \mid P(f)\ \} \tag{5.3}$$

All entries in U also exist in E, making U a subset of E itself. Such a statement could not be made if $|F| \geq 0$ instead of $|F| > 0$. By performing the query plan transformation to replace the `SCAN` operator with a secondary index `SEARCH` and primary index `SEARCH`, we return E to the rest of the query plan. We can take advantage of the existing secondary index validation step to remove all entries $e \in E$ that do not exist in U. Such an approach returns the correct results back to the user and benefits from the use of the index.

5.5.4 Join Queries with Arrays in the Join Predicate

Figure 5.11 describes a non-indexed query plan for the join queries in the previous chapter (Listing 4.29 and Listing 4.30). Two `SCAN` operators exist here: one for the outer join dataset `Products` (producing R records) and one for the inner join dataset `Orders` (producing S records). On the left branch (the outer dataset), all records R must satisfy the predicate on R.`name` before the qualifying R.`product_id` values are given to the `JOIN` operator. On

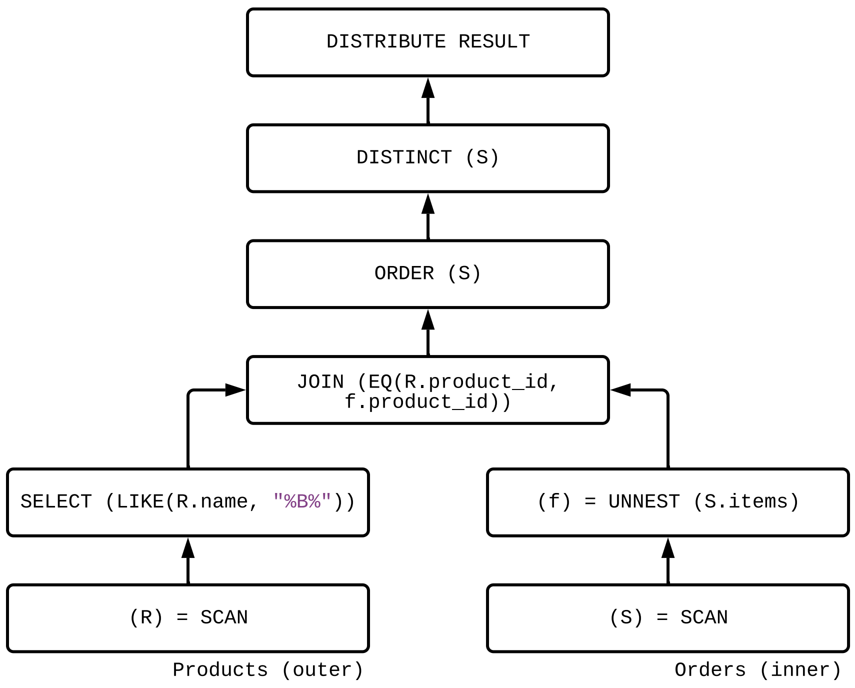


Figure 5.11: Example two-dataset join query plan.

the right branch (the inner dataset), the S.`items` array goes through an `UNNEST` operator and the `product_id` field is given to the `JOIN` operator. To satisfy the `DISTINCT` clause, the qualifying S records after the join are given to an `ORDER` operator and `DISTINCT` operator before being given to the user.

Figure 5.12 describes the transformed query plan that utilizes a multi-valued index to perform an *index-nested loop join*. For an outer dataset with records R and an inner dataset with records S, with an index on the join field for the inner dataset, an index-nested loop join performs an index search on the join field for each record R. The transformed query plan starts with the `SCAN` operator and the `SELECT` operator on the `Products` dataset to retrieve the probe records R to join. The join field R.`product_id` is then used to search the applicable multi-valued index to return the primary key for any matching S records (S.`order_id`). Duplicate S.`order_id` values are removed with the `ORDER` and `DISTINCT` operators before being given to the primary index `SEARCH` to retrieve the records S. Secondary index validation is performed by extracting the atomic endpoints and placing the join predicate in a `SELECT`

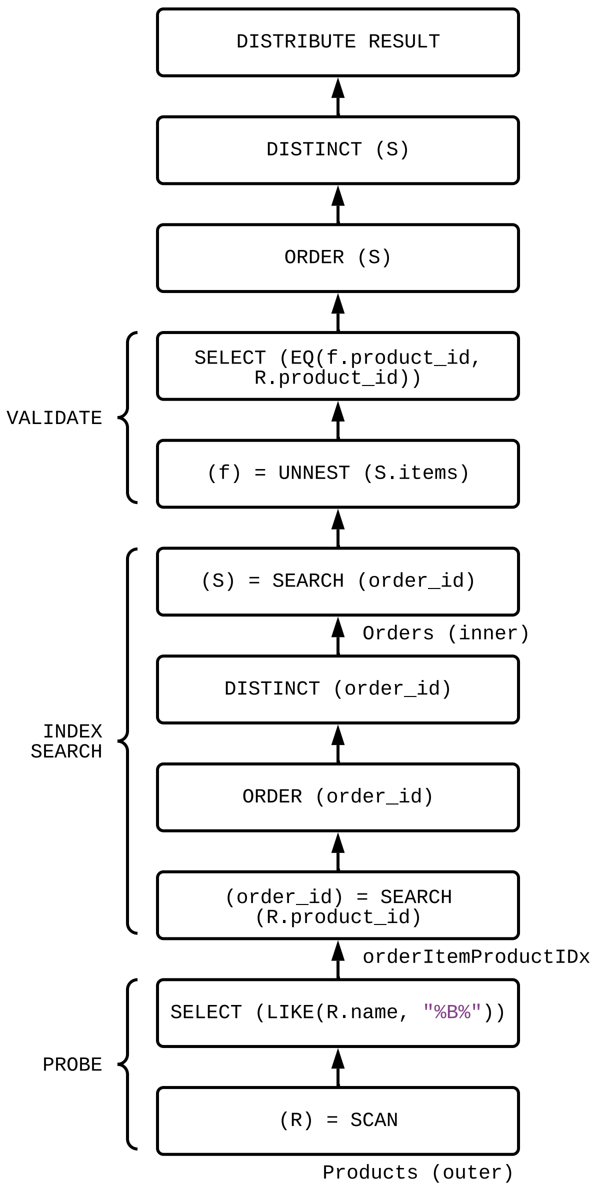


Figure 5.12: Example index-nested loop join query plan.

operator. Finally to satisfy the query's DISTINCT clause, the records that survived the validation step are given to another set of ORDER and DISTINCT operators before being given to the user.

Similar criteria from Subsection 5.5.1 are used to recognize index applicability:

1. Given some predicate in a JOIN operator for a query plan, there must exist some conjunct C in the aforementioned predicate.

2. C must be the `EQ` function.
3. The sequence of `UNNEST` clauses and associated paths in some index specification must match the sequence of `UNNEST` operators in inner dataset's join branch.
4. The argument to the function C must be a path specified in the `SELECT` clause of the index specification or the output of the `UNNEST` operator if no `SELECT` clause exists.

Potential future work includes supporting join queries beyond explicit `UNNEST` clauses, such as joins through existential quantification.

Chapter 6

Evaluation

This chapter details two types of experiments. The first is an experiment to characterize the maintenance overhead associated with the new AsterixDB implementation of multi-valued indexes, and the second is to demonstrate the efficacy of multi-valued indexes for analytical queries in AsterixDB.

6.1 Atomic vs. Single-Item Array Index Maintenance

Multi-valued indexes can massively accelerate select queries, but it is important to understand the cost associated with maintaining such an index. For every maintenance operation applied to a dataset's primary index, the same type of maintenance must also be applied to all secondary indexes on said dataset. In this section, we characterize such operations for multi-valued indexes by comparing their execution times against AsterixDB's existing atomic B+ tree indexes.

6.1.1 Experimental Setup

All experiment runs were performed on a single-node AsterixDB instance, executed on an Intel Celeron J4125, 4 cores @ 2.7GHz CPU with 8GB of RAM and a single NGFF M.2 SSD. To normalize the cost associated with each storage layer write, multi-valued indexes in this experiment are restricted to a *single* item. This experiment can also be thought of as comparing the cost of performing `INSERT`, `DELETE`, and `UPSERT` operations for an atomic element wrapped in an array. Three `ShopALot` datasets with only single-item arrays were generated and used for comparison: the `Users` dataset, the `Stores` dataset, and the `Orders` dataset. All datasets used were larger than memory (`Users` being 15GB @ 100 million records, `Stores` being 20GB @ 90 million records, and `Orders` being 18GB @ 65 million records), resulting in indexes that were larger than memory as well. One index was created on the `number` field inside the `phones` object array of the `Users` dataset, with each phone number being unique from one another. One index was created on the `categories` string array of the `Stores` dataset, with each category randomly sampled from a list of 15 different values. Two indexes were created on the `Orders` dataset, one on the `qty` field (the absolute value of an integer normally distributed with mean 1 and deviation 10) inside the `items` object array and another on the `product_id` field (a string randomly sampled from a list of 541 different values) inside the same `items` object array. For `INSERT` and `UPSERT` experiment instances, 10,000 record chunks at a time were used until the dataset grew 0.5% in size. For `DELETE` experiment instances, 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, an atomic index on a 10,000 record chunk identifier was created and used.

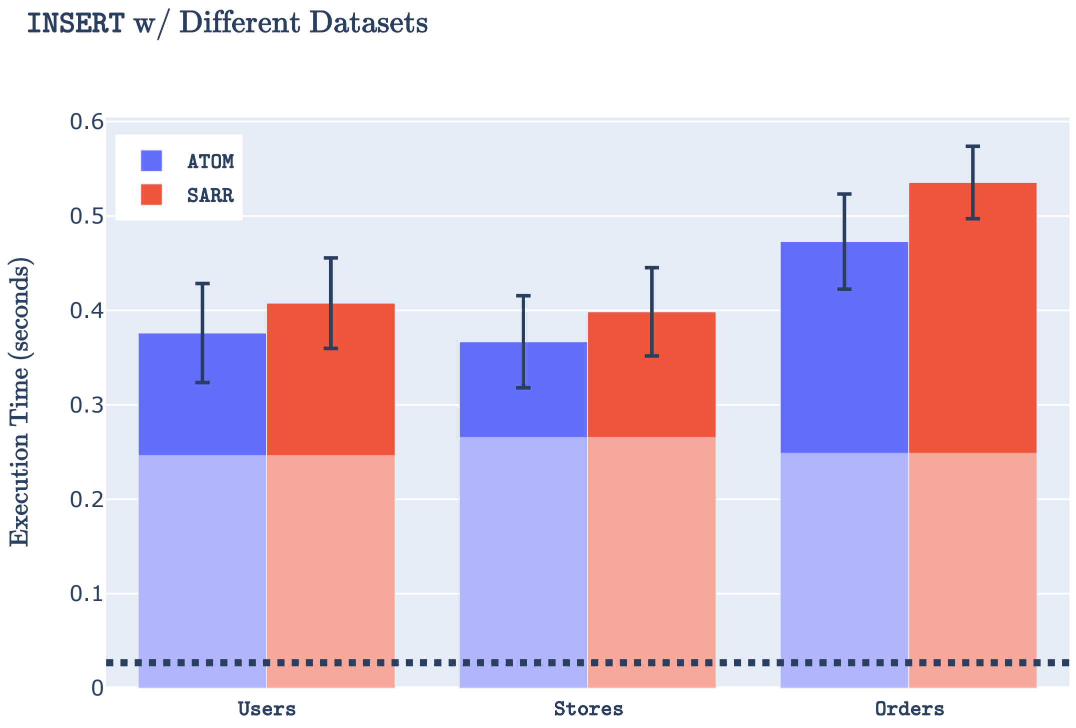


Figure 6.1: Execution times across different datasets for INSERT statements.

6.1.2 Results

Figure 6.1 displays the average time to perform 10,000-record INSERT statements for the three datasets mentioned previously. 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 portions of the bar graph represent the time to perform the INSERT statement on a non-indexed dataset (this time is similar for both datasets). There does exist a slight slowdown for multi-valued indexes when compared to atomic indexes, most emphasized in the dataset with two indexes (i.e. Orders). This small difference comes from the inclusion of the extra operators to extract the secondary key values and the overhead of using subplans.

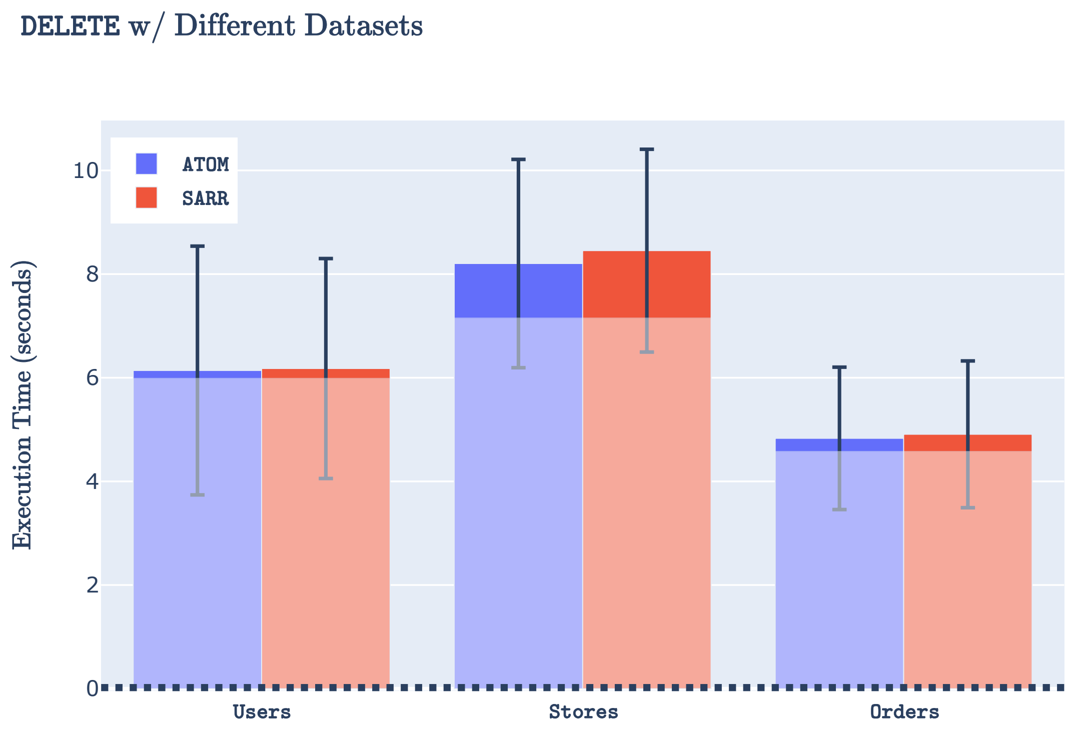


Figure 6.2: Execution times across different datasets for `DELETE` statements.

Figure 6.2 displays the average time to perform the 10,000-record `DELETE` statements for the same three 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 all `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 atomic 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 this observation. The cost of performing a record deletion for some 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 [1].

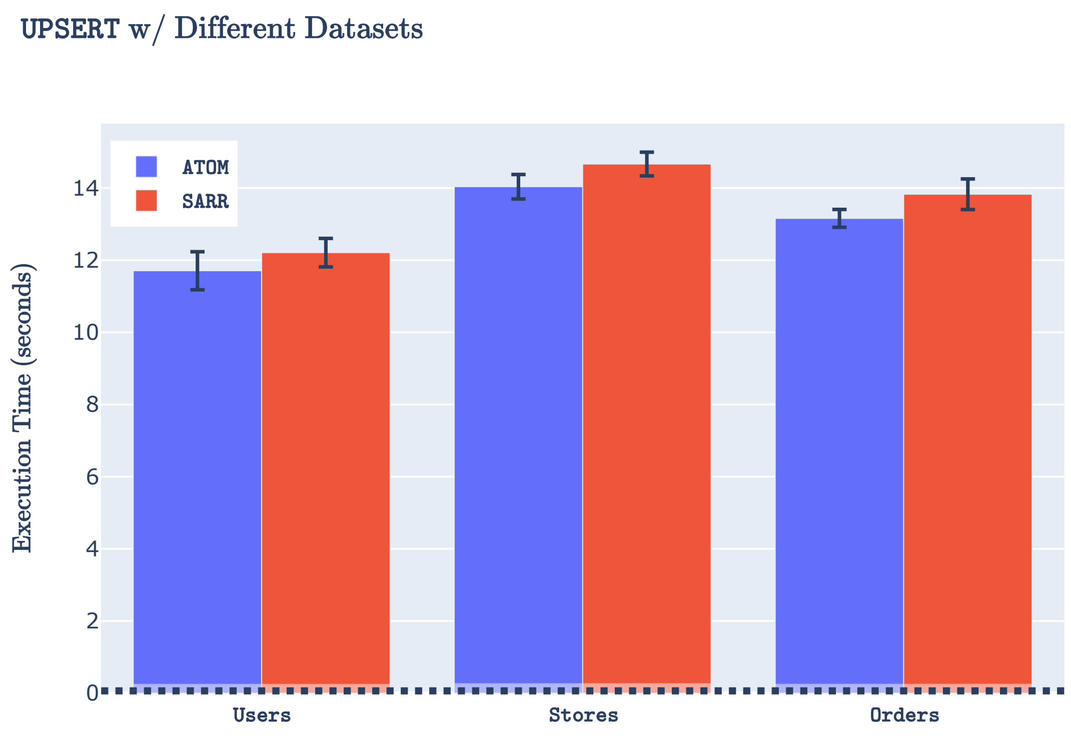


Figure 6.3: Execution times across different datasets for UPSERT statements.

Figure 6.3 describes the average time to perform a 10,000-record UPSERT statement 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. UPSERT statements generally take longer to execute than DELETE statements, due to the fact that the primary keys used to search the corresponding dataset are *not* sorted beforehand. As noted in [14], 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. If the records to-be-fetched are located far from each other, then DELETE will perform on-par with UPSERT. Such an observation explains the large deviation in `DELETE` times but not `UPSERT` times. The large difference in `UPSERT` times between the indexed and non-indexed datasets (i.e. the opaque and solid bars) is due to an optimization AsterixDB takes in the presence of no secondary indexes (the opaque bars). If there are no secondary indexes on a dataset, the output from a primary index `UPSERT` operator (i.e. the old record 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 this `UPSERT` operation.

6.2 Queries with Multi-Valued Indexes

To demonstrate the effectiveness of multi-valued indexes in an analytical setup, this section compares the execution time of a set of benchmark queries with and without a multi-valued index.

6.2.1 Experimental Setup

All experimental runs were performed on a single-node AsterixDB instance, executed on an Intel Celeron J4125, 4 cores @ 2.7GHz CPU with 8GB of RAM and a single NGFF M.2 SSD. The benchmark used here is a modified version of the CH benchmark, which is a combination of TPC-C and TPC-H for a HOAP (hybrid operational / analytical processing) workload [8]. CH originally assumes a flat relational data model, which does not utilize the richer features that a document data model supports. Thus a more natural document-oriented collection of datasets were utilized for this experiment (inspired by [28, 15]). In particular the `OrderLine` entity, a weak entity attached to the `Orders` dataset, was translated as an array within inside the `Orders` dataset itself (resembling the `items` field in the `ShopALot` `Orders` dataset). This `Orders` dataset is described with the complete type definition in Listing 6.1.

```
CREATE TYPE OrdersType AS {
    o_id: int,
    o_d_id: int,
    o_w_id: int,
    o_c_id: int,
    o_entry_d: string,
    o_carrier_id: int,
    o_ol_cnt: int,
    o_all_local: int,
    o_orderline: [{
        ol_i_id: int
        ol_number: int,
        ol_supply_w_id: int,
        ol_delivery_d: string,
        ol_quantity: int,
        ol_amount: float,
        ol_dist_info: string
    }]
};
```

Listing 6.1: Complete type specification for the CH `Orders` dataset.

The AsterixDB type definitions for each dataset in each experimental run retained their original primary keys, but all other fields were omitted from their type definitions to more realistically model the schemaless-ness of a typical NoSQL environment. A total of 200 CH warehouses were generated for this experiment, resulting in two datasets larger than memory: `Orders` (at 11GB) and `Stock` (at 11GB). One multi-valued index was built on the `ol_delivery_d` field inside the `o_orderline` object array of the `Orders` dataset.

6.2.2 Results

Queries in this section are split into two sets: (i) queries that do not involve a join with the other larger-than-memory dataset `Stock`, and (ii) queries that involve a join with the `Stock` dataset. All queries in each dataset can be found in Appendix A. Figure 6.4 details the execution times of the former queries as a function of σ, the selectivity of the index-applicable predicate. σ is varied between 0% and 100%, with the lower σ values yielding few records from the dataset that the index is on. Indexed queries achieve sub-second execution times with $\sigma = 0.01$, while the same queries without an index at $\sigma = 0.01$ consistently run longer than 3 minutes. As σ grows larger and larger though, the execution times for the indexed queries grow faster than for non-indexed queries. Eventually the query plan integrating multi-valued indexes is not the fastest data flow. A plan that utilizes a secondary index search followed by a primary index search is vastly superior in execution times when the applicable query predicate has a low selectivity [14]. At $\sigma \gtrapprox 20$, the indexed execution time becomes worse than a plan that does not utilize the secondary index, as the process of searching the primary index for each qualifying secondary index entry becomes more and more expensive. In the worst case (if every single secondary index entry were to qualify), then a primary index search would have to be performed for each and every entry, which is much slower than just performing a scan over the primary index.

Figure 6.5 tells a similar story with a different query set, the join set, though the difference in execution times between their indexed and non-indexed queries is smaller. The reason for the smaller difference is because each resulting record of `Orders` dataset is eventually joined with all qualifying records of the large dataset `Stock`. This join leads to a higher floor than the previous query set, though a small selectivity σ in conjunction with an applicable index does result in faster join query execution times.

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. Just as with atomic 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.

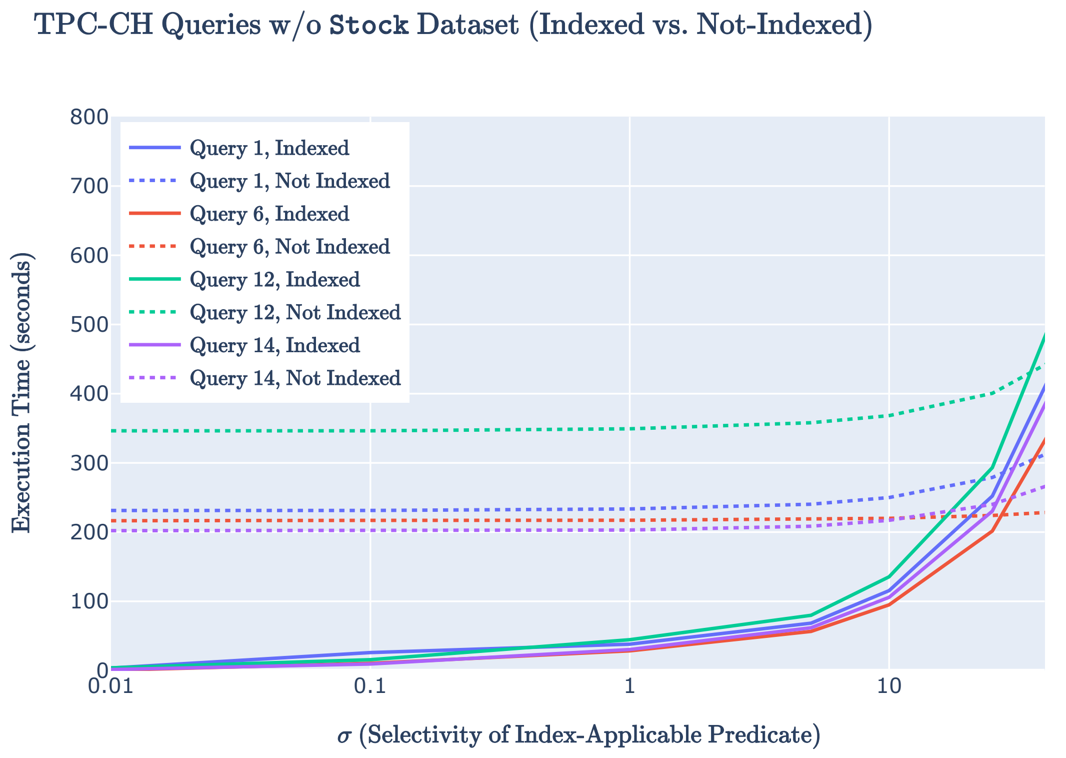


Figure 6.4: Indexed vs. not-indexed query execution times for query set 1.

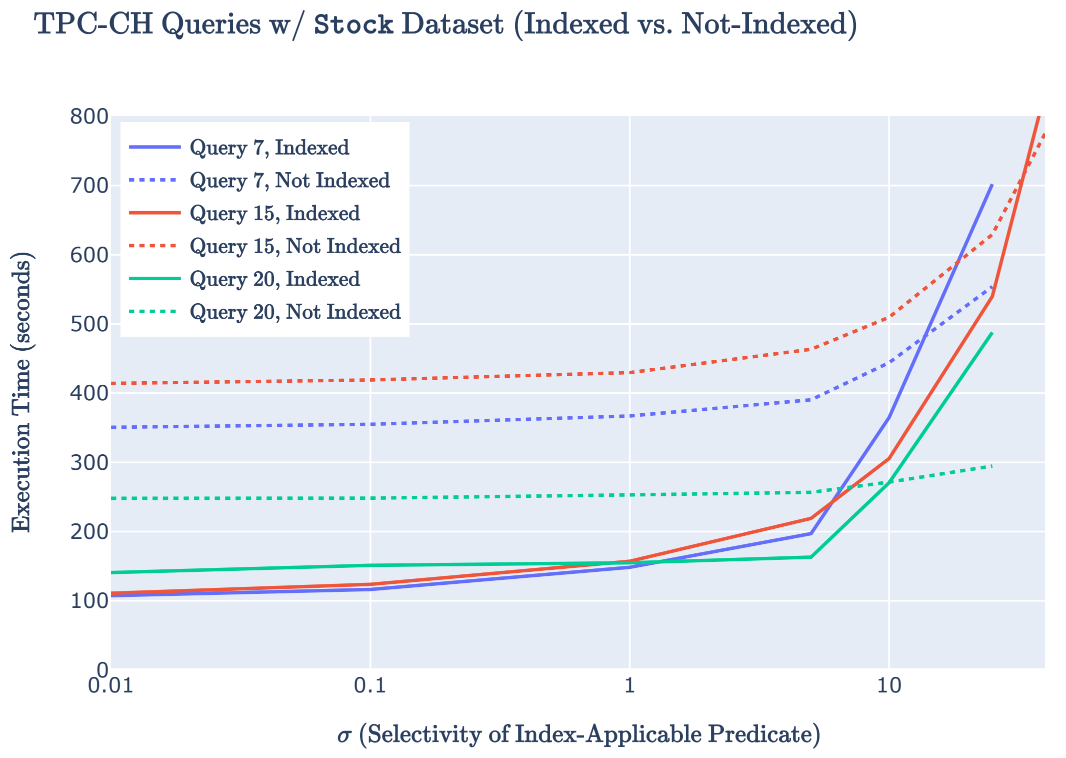


Figure 6.5: Indexed vs. not-indexed query execution times for query set 2.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis has described the various steps needed to realize multi-valued indexes in AsterixDB. We started by describing the indexing user experience: (i) How multi-valued indexes should be specified, justified by requirements we developed throughout each subsection, and (ii) what queries should be accelerated by a multi-valued index. We then unraveled the indexing implementation, detailing what a single multi-valued index entry represents and demonstrating different data flows to maintain and utilize multi-valued indexes. We concluded by characterizing the maintenance overhead associated with multi-valued indexes and demonstrated the tremendous efficacy of multi-valued indexes for analytical benchmark queries with low selectivity on the index.

7.2 Potential Future Work

In this section, we will discuss four pieces of potential future work: (i) implementing composite atomic + multi-valued indexes, (ii) revising the UPSERT logic for multi-valued indexes, (iii) comparing multi-valued indexes in AsterixDB with other implementations of multi-valued indexes in different systems, and (iv) realizing multi-valued R Tree indexes.

7.2.1 Composite Atomic + Multi-Valued Indexes

To remind the reader, recall one of the example index specifications given in Subsection 4.2.4:

```
CREATE INDEX storeZipCatIdx ON Stores (
    address.zip_code,
    UNNEST categories
);
```

Listing 7.1: Example composite index specification with an atomic element prefix and a multi-valued element suffix (same as Listing 4.19).

While composite atomic + multi-valued index construction and maintenance is currently supported, utilizing such indexes to accelerate queries is not. Continuing from the index specification in Section 4.2.4, with a composite index on the zip_code field inside the address object field and the categories string array of the Stores dataset, the following query could then be optimized:

```
FROM      Stores S
UNNEST    S.categories C
WHERE     S.address.zip_code = "92617" AND
          C = "Produce"
SELECT    DISTINCT S;
```

Listing 7.2: Example query that could benefit from the previous composite atomic multi-valued index.

In contrast to the query plans presented in Section 5.5, the predicates that we need to search for to determine index applicability would be split across more than one `SELECT` operator. (In AsterixDB, we currently only search for index applicability within a *single* `SELECT` operator.)

7.2.2 Revised Upsert Logic

Recall that in Subsection 5.4.4, the current approach to perform an `UPSERT` operation on a multi-valued index is to *unconditionally* delete all entries from the multi-valued field of the old record and insert all entries from the multi-valued field of the new record. If the indexed multi-valued field from the record to-be-`UPSERT`ed has not changed, then we will perform a slew of unnecessary deletions and insertions. This extraneous work becomes more egregious with larger arrays and multisets. Atomic indexes in AsterixDB, on the other hand, avoid these extra storage layer invocations (delete and insert) if the value of the indexed field for the old record is the same as the value of the indexed field for the new record. Replicating the same `UPSERT` logic for multi-valued indexes presents the challenge of essentially comparing two multi-valued fields using two distinct subplans.

7.2.3 AsterixDB vs. Other Systems

AsterixDB is not the only document database system to implement multi-valued indexing. As described in Chapter 2, document database systems like MongoDB, Couchbase, and Oracle NoSQL each now offer their own flavor of multi-valued indexes. Couchbase in particular offers two interesting multi-valued index types: (i) array indexes, which utilize B+ trees for storage, and (ii) flex indexes, which utilize inverted indexes for storage. Modern relational stores such as MySQL offer some multi-valued indexing on JSON arrays as well. It would be interesting to compare the performance across these systems, given their different design decisions, for the same analytical queries in Section 6.2.

7.2.4 R Tree Indexes

As mentioned in Section 4.3, the work performed in this thesis is general enough to be applied to RTREE indexes as well. Given a document containing a collection of spatial data (e.g. two-dimensional points), certain spatial queries could be accelerated by building upon the foundation laid here. Suppose that some records in the Users dataset had an array of two-dimensional points that indicate their preferred locations (denoted as preferred_locations). Now assume that an application issues the query below to search for all users that have a preferred location within some rectangular space.

```
WITH      SearchRectangle AS CREATE_RECTANGLE (
              CREATE_POINT(30.0, 70.0),
              CREATE_POINT(40.0, 80.0)
          )
FROM      Users U
WHERE     SOME P IN U.preferred_locations
          SATISFIES SPATIAL_INTERSECT(P, SearchRectangle))
SELECT    U;
```

Listing 7.3: Example query that could benefit from a multi-valued R Tree index.

Supporting R Tree indexes would require effort to connect the existing R Tree index work (the bulk-loading, the index maintenance, and the query optimization) to this multi-valued index work within the SQL++ compiler and Job Execution components of AsterixDB.

Bibliography

[1] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in AsterixDB. *Proc. VLDB Endow.*, 7(10):841–852, June 2014.

[2] AsterixDB. Apache AsterixDB, a scalable open source big data management system (BDMS). Available at `https://asterixdb.apache.org`, 2021.

[3] E. Bertino and P. Foscoli. Index organizations for object-oriented database systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):193–209, 1995.

[4] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):196–214, 1989.

[5] V. Borkar, Y. Bu, E. P. Carman, N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey, and 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, page 422–433, New York, NY, USA, 2015. Association for Computing Machinery.

[6] V. Borkar, M. Carey, R. Grover, N. Onose, and 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, page 1151–1162, USA, 2011. IEEE Computer Society.

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

[8] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload CH-BenCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, DBTest '11, New York, NY, USA, 2011. Association for Computing Machinery.

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

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

[11] O. Deux. The story of O2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.

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

[13] 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)*, pages 47–54, 1997.

[14] G. Graefe. Modern B-Tree techniques. *Found. Trends Databases*, 3(4):203–402, Apr. 2011.

[15] A. Kamsky. Adapting TPC-C benchmark to measure performance of multi-document transactions in MongoDB. *Proc. VLDB Endow.*, 12(12):2254–2262, Aug. 2019.

[16] A. Kemper and G. Moerkotte. Access support in object bases. *SIGMOD Rec.*, 19(2):364–374, May 1990.

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

[18] Y.-S. Kim. *Transactional and Spatial Query Processing in the Big Data Era.* PhD thesis, University of California, Irvine, 2016.

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

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

[21] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented DBMS. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '86, page 472–482, New York, NY, USA, 1986. Association for Computing Machinery.

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

[23] MySQL. Wl 8955: Add support for multi-valued indexes. Available at `https://dev.mysql.com/worklog/task/?id=8955#tabs-8955-4`, 2021.

[24] M. Nicola and 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, and B. C. Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 1164–1174. ACM, 2005.

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

[26] Y. Papakonstantinou, A. Goo, B. Ruppert, J. Wilsdon, and P. Varakur. Announcing PartiQL: One query language for all your data. Available at https://aws.amazon.com/blogs/opensource/announcing-partiql-one-query-language-for-all-your-data, 2019.

[27] T. Sigaev and O. Bartunov. Gin for postgresql. Available at http://www.sai.msu.su/~megera/wiki/Gin, 2008.

[28] Y. Tian, M. Carey, and I. Maxon. Benchmarking HOAP for scalable document data management: A first step. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 2833–2842, 2020.

[29] P. Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, June 1987.

Appendix A

Benchmark Queries

This appendix describes the two query sets [8] used for the experiments in Section 6.2. To modify the selectivity for each experiment, the dates "D1" and "D2" in each query below were varied.

A.1 Query Set 1: Without Stock Join

A.1.1 CH Benchmark Query 1

Report the total amount and quantity of all shipped order-lines within a specific time period. Additionally report the average amount and quantity, and the total count of all order-lines ordered by the individual order-line number.

```
FROM         Orders O, O.o_orderline OL
WHERE        OL.ol_delivery_d BETWEEN "D1" AND "D2"
GROUP BY     OL.ol_number
SELECT       OL.ol_number,
             SUM(OL.ol_quantity) AS sum_qty,
             SUM(OL.ol_amount) AS sum_amount,
             AVG(OL.ol_quantity) AS avg_qty,
             AVG(OL.ol_amount) AS avg_amount,
             COUNT(*) AS count_order
ORDER BY    OL.ol_number;
```

A.1.2 CH Benchmark Query 6

List the total amount of archived revenue from order-lines that were delivered in a specific period and a certain quantity.

```
FROM     Orders O, O.o_orderline OL
WHERE    OL.ol_delivery_d BETWEEN "D1" AND "D2" AND
         OL.ol_quantity BETWEEN 1 AND 100000
SELECT   SUM(OL.ol_amount) AS revenue;
```

A.1.3 CH Benchmark Query 12

Count the number of high and low priority orders, grouped by the number of order-lines in each order.

```
FROM        Orders O, O.o_orderline OL
WHERE       O.o_entry_d <= OL.ol_delivery_d AND
            OL.ol_delivery_d BETWEEN "D1" AND "D2"
GROUP BY    O.o_ol_cnt
SELECT      O.o_ol_cnt,
            SUM(CASE WHEN O.o_carrier_id = 1 OR
                          O.o_carrier_id = 2
                THEN 1 ELSE 0 END) AS high_line_count,
            SUM(CASE WHEN O.o_carrier_id <> 1 OR
                          O.o_carrier_id <> 2
                THEN 1 ELSE 0 END) AS low_line_count
ORDER BY    O.o_ol_cnt;
```

A.1.4 CH Benchmark Query 14

Get the percentage of the revenue in a period of time which has been realized from promotional campaigns.

```
FROM     Item I, Orders O, O.o_orderline OL
WHERE    OL.ol_i_id = I.i_id AND
         OL.ol_delivery_d BETWEEN "D1" AND "D2"
SELECT   100.00 * SUM(CASE WHEN I.i_data LIKE 'pr%'
                      THEN OL.ol_amount ELSE 0 END) /
             (1 + SUM(OL.ol_amount)) AS promo_revenue
```

A.2 Query Set 2: With Stock Join

A.2.1 CH Benchmark Query 7

Show the bi-directional trade volume between two given nations sorted by their names and the considered years.

```
FROM         Supplier SU, Stock S, Orders O,
             O.o_orderline OL, Customer C,
             Nation N1, Nation N2
WHERE        OL.ol_supply_w_id = S.s_w_id AND
             OL.ol_i_id = S.s_i_id AND
             ((S.s_w_id * S.s_i_id) % 10000) = SU.su_suppkey AND
             C.c_id = O.o_c_id AND
             C.c_w_id = O.o_w_id AND
             C.c_d_id = O.o_d_id AND
             SU.su_nationkey = N1.n_nationkey AND
             STRING_TO_CODEPOINT(SUBSTR(C.c_state, 1, 1))[0]
                 = N2.n_nationkey AND
             ( ( N1.n_name = 'Germany' AND
                 N2.n_name = 'Cambodia' ) OR
               ( N1.n_name = 'Cambodia' AND
                 N2.n_name = 'Germany' ) ) AND
             OL.ol_delivery_d BETWEEN "D1" AND "D2"
```

```
GROUP BY     SU.su_nationkey,
             STRING_TO_CODEPOINT(SUBSTR(C.c_state, 1, 1))[0],
             SUBSTR(O.o_entry_d, 0, 4)
SELECT       SU.su_nationkey AS supp_nation,
             STRING_TO_CODEPOINT(SUBSTR(C.c_state, 1, 1))[0]
                 AS cust_nation,
             SUBSTR(O.o_entry_d, 0, 4) AS l_year,
             SUM(OL.ol_amount) AS revenue
ORDER BY     SU.su_nationkey, cust_nation, l_year;
```

A.2.2 CH Benchmark Query 15

Find the top supplier or suppliers who contributed the most to the overall revenue for items shipped during a given period of time.

```
WITH         Revenue AS (
    FROM         Stock S, Orders O, O.o_orderline OL
    WHERE        OL.ol_i_id = S.s_i_id AND
                 OL.ol_supply_w_id = S.s_w_id AND
                 OL.ol_delivery_d BETWEEN "D1" AND "D2"
    GROUP BY     ((S.s_w_id * S.s_i_id) % 10000)
    SELECT       ((S.s_w_id * S.s_i_id) % 10000) AS supplier_no,
                 SUM(OL.ol_amount) AS total_revenue
)
FROM         Supplier SU, Revenue R
WHERE        SU.su_suppkey = R.supplier_no AND
             R.total_revenue = (
                 FROM    Revenue
                 SELECT  VALUE MAX(total_revenue)
             )[0]
SELECT       SU.su_suppkey,
             SU.su_name,
             SU.su_address,
             SU.su_phone,
             R.total_revenue
ORDER BY     SU.su_suppkey;
```

A.2.3 CH Benchmark Query 20

Find suppliers in a nation having selected parts 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 since a certain date.

```
FROM         Supplier SU, Nation N
WHERE        SU.su_suppkey IN (
    FROM         Stock S, Orders O, O.o_orderline OL
    WHERE        S.s_i_id IN (
                     FROM     Item I
                     WHERE    I.i_data LIKE 'co%'
                     SELECT   VALUE I.i_id
                 ) AND
                 OL.ol_i_id = S.s_i_id AND
                 OL.ol_delivery_d BETWEEN "D1" AND "D2"
    GROUP BY     S.s_i_id, S.s_w_id, S.s_quantity
    HAVING       (100 * S.s_quantity) >
                     SUM(OL.ol_quantity)
    SELECT       VALUE ((S.s_w_id * S.s_i_id) % 10000)
             ) AND
             SU.su_nationkey = N.n_nationkey AND
             N.n_name = 'Germany'
SELECT       SU.su_name,
             SU.su_address
ORDER BY     SU.su_name;
```

Appendix B

Issues with Multi-Valued Index-Only-Scan Queries

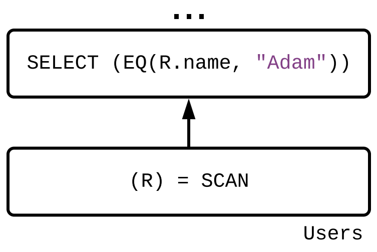
As mentioned in Section 5.2, covering queries, or *index-only-scan* queries, are not supported for use with multi-valued indexes at the time of writing. This appendix explains the challenges that led to that decision.

Recall the discussion with respect to locking in Subsection 5.4.1: All entries retrieved from a secondary index are validated by retrieving the referenced entry from the corresponding primary index because no locks are placed on secondary indexes themselves. This validation is demonstrated with the query plan snippets in Figure B.1 for the equality query in Listing B.1 on the `Users` dataset.

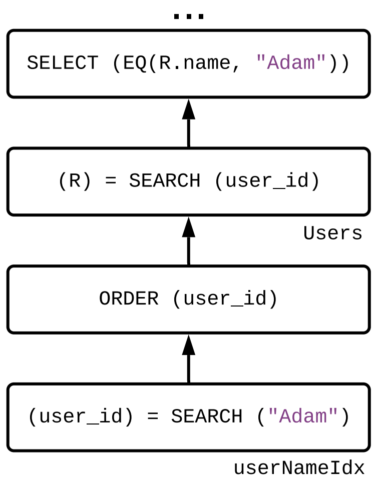
```
FROM     Users U
WHERE    U.name = "Adam"
SELECT   *;
```

Listing B.1: Example query to illustrate index acceleration with the query plan in Figure B.1.

If there exists no index on the `name` field, then the query plan on the left Figure B.1a will be



(a) Example query plan without secondary index.



(b) Example query plan with secondary index.

Figure B.1: Partial example query plans to demonstrate secondary index validation.

executed: (i) A full dataset scan is performed on the `Users` dataset. (ii) A filter is performed on each record that satisfies the equality predicate R.`name` = `"Adam"` for all records in R.

If there does exist an index on `name` field, then the query plan on the right will be executed: (i) A B+ tree search is performed on the aforementioned index (`userNameIdx`) for all entries with `name` equal to `"Adam"`. This returns all qualifying index entries, with each entry consisting of the sort key (`name`) and the primary key (`user_id`). (ii) The primary key field (`user_id`) is used to perform another B+ tree search using the primary index to retrieve the qualifying records R themselves. Note that for efficient navigation of the primary index [14], all fetched primary keys undergo a sort operation before the primary keys themselves are given to the search operator. (iii) The secondary index validation is performed with the final operator presented in this plan, the `SELECT` operator. Index-only-scan plans in AsterixDB work around the limitation of not locking secondary indexes by first checking if a primary index lock could be successfully requested on a retrieved secondary index entry. If this lock could be granted, then no other transactions are currently modifying the retrieved secondary index entry. Consequently there is no need to validate this entry and a primary index search can be skipped altogether. If this lock is not available, then validation is required and a primary index search for this specific entry is performed. This split sequence of operations is reflected in Figure B.2. The lock decision is performed in the `SPLIT` operator, and searched index entries are merged back together in the `UNION` operator.

AsterixDB index-only-scan plans are currently executed using record-level locks for each index *entry*. Suppose that index entries were returned from a multi-valued index B+ tree search. Primary key values can now map to more than one index entry, meaning that the check to see if a primary index lock can be requested is now performed for each item in the multi-valued field. Multiple checks means that entries with the same primary key can traverse different branches of the `SPLIT` operator, resulting in the appareance of multi-valued fields (i.e. value sets) that may have never existed. We can see this via the following hypothetical scenario for a multi-valued index using the same index-only-scan plan for atomic indexes in Figure B.2:

1. Suppose that we are working with a multi-valued index on the `categories` array of the `Stores` dataset. `Stores` in this situation holds one record with the primary key value `"D1"` and yields two index entries: (i) `("Bread", "D1")`, and (ii) `("Deli", "D1")`
2. A B+ tree search is performed on `storesCatIdx` and both index entries are retrieved.
3. Now suppose that a primary index lock can be successfully requested for the record `"D1"`, so the `("Bread", "D1")` entry traverses the right branch of the index-only-scan query plan and does not have to retrieve a record from the primary index.
4. Concurrently, a `DELETE` is called on the `"D1"` record itself. The primary index lock request check will now fail for `"D1"`, so the `("Deli", "D1")` entry traverses the left branch of the index-only-scan query plan. After the other transaction's `DELETE` finishes, a lock can be acquired on the left branch on the primary index to search for the `"D1"` record. No such record is found, and no records are returned from this primary index search.

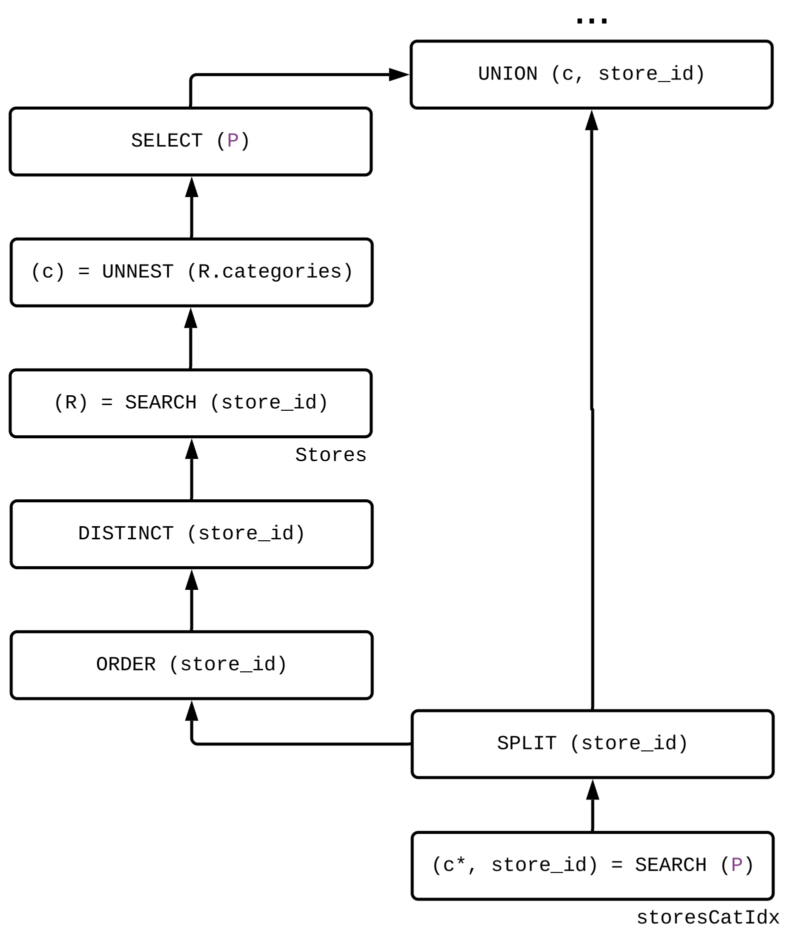


Figure B.2: Example hypothetical partial index-only-scan query plan for a multi-valued index to demonstrate the locking issue.

5. The sole entry `("Bread", "D1")` would now be incorrectly given to the rest of the query plan. An invalid state is demonstrated here, as the only valid states with respect to the `"D1"` record are either both entries `("Bread", "D1")` and `("Deli", "D1")` or no entries at all. At no point in time in this scenario did a record with the primary key `"D1"` exist with only one item in its `categories` array.

Though this locking issue is not the only problem with multi-valued index-only-scan query plans (another issue is how to handle the `UNION` operation correctly), it is the largest issue to be overcome to realize covering multi-valued indexes.