Computer
Sciences
Department

Efficient Storage and Query Processing of Set-Valued Attributes

Karthikeyan Ramasamy

Technical Report #1436

February 2002

UNIVERSITY OF
WISCONSIN
MADISON

EFFICIENT STORAGE AND QUERY PROCESSING OF SET-VALUED ATTRIBUTES

By

Karthikeyan Ramasamy

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

(COMPUTER SCIENCES)

at the

UNIVERSITY OF WISCONSIN – MADISON

2001

Abstract

In order to better support complex applications, object relational systems provide features that are absent in relational systems. The main distinguishing features are abstract data types and type constructors. The set constructor creates a new type by providing collections of existing types. It is well known that sets are useful in modeling a great deal of real world data. However, such powerful modelling comes at a price; without an efficient implementation, using sets can yield a performance much worse than that obtained using only traditional relational constructs. This dissertation explores novel ways of implementing set-valued attributes in an object relational system. Specifically, it considers various options for efficiently storing set-valued attributes, and ways of computing the challenging set containment join operation.

We first address the problem of storing set-valued attributes. Using the orthogonal attributes of nesting and location we identify four options for representing sets: nested internal and external, and unnested external and internal. These representations can be combined with the creation of various indices to create various classes of indexed representations. We evaluate each of these representations with respect to conjunctive and disjunctive queries. Our results show that overall the nested implementations perform better than the unnested implementations because (a) they exploit grouping semantics while fetching the members of a set instance and (b) they allow the evaluation of set predicates directly on the set instance.

Next we consider the problem of efficiently evaluating set containment joins. For unnested external representation, the set containment join can be expressed directly in SQL. By contrast, the most obvious algorithm for computing set containment joins on nested representations is the signature nested loops algorithm, which computes set signatures and compares each signature in a relation with all the signatures in the other relation. To improve on the performance of this algorithm we propose a new partitioned set join algorithm (PSJ), which uses a multi-level scheme of partitioning by replicating the inner relation. Our performance study shows that for extremely small relation and small set cardinalities, the SQL query approach and signature nested loops perform comparably to PSJ. However, as the size of the data sets increase (in both relation and set cardinality), PSJ clearly dominates.

Acknowledgments

First, I would like to thank my advisor Jeff Naughton. He has been a great source of inspiration and role model to me. This research would not have been completed without his crystal clear thoughts and insights when the problems got really tough. It is amazing how fast he can adapt and present the research results in an organized manner. Some day, I would love to emulate his writing and presenting skills. It has been really great working with him.

Next, I would like to thank David Dewitt for introducing me to Paradise. Interacting with David has been a rewarding experience for me and significantly improved my thinking about system building. I will never the forget the e-mail that he sent when I finished porting of Paradise to SGI platforms in 3 days. I would like to thank Raghu Ramakrishnan and Yannis Ioannidis who generously shared their wider perspective in the field of databases.

Next, I would like to thank Mike Carey for introducing me to set-valued attributes when I was working with him in the OQL project. This ultimately led to the thesis topic. He has been a constant source of advice for me whenever needed. I am grateful to David Maier, Jignesh Patel and Raghav Kaushik for providing valuable insights that significantly changed the course of my research.

Working with Jiebing Yu, Jignesh Patel, Navin Kabra, Biswadeep Nag, Kristin Tufte, Josef Burger, Jim Kupsch, Curt Ellman and Roger Lueder of the Paradise group has been enriching. This lead to my deeper understanding of architecting large scale systems. Thanks to SHORE group members Mike Zwilling and Nancy Hall for teaching the internals of SHORE that helped further shape my system skills.

Initiating the BORG project and leading to completion has been really a very good experience for me. Working with Prasad Deshpande and Amit Shukla in has been really enriching. It was excellent working with Rick Stellwagen, CTO of NCR (a person with unbounded amount of energy) incorporating ideas of BORG into NCR products.

During my tenure as a graduate student, it was a great learning experience having database related discussions with Kevin Beyer, Venkatesh Ganti, Jayavel Shanmugasundaram, Johannes Gehrke, Natassa Ailamaki, Donko Donjerkovic, Vishy Poosala and Praveen Shasadri.

Life would not have been wonderful during the past seven years without the following people:

- Thanks to Meeta and Navin Kabra for inviting often for dinner and providing entertainment with trivia games.
- Thanks to Prasad Deshpande for putting with me. We had a wonderful time cross country driving from Madison to San Diego.
- Thanks to Deepa Seshan and Maharshi Chauhan who wake me up at 3.00 am in the morning from Chicago and remind me it is time to work :)) It has been great fun.
- Thanks to Preeti Dubey and Bhalachandra Puranik whom I never believed will end up together. It is analogous to a cat and a dog getting married :)))
- Thanks to Shruti Mukhtyar and Amit Shukla for their fun company to movies and dinners.
- Thanks to Venkatesh Iyengar (@Brain) for his "greatness" :))

- Thanks to Pawan Joshi and Anand Krishnamurthy for their wonderful company in Madison.

- Thanks to Neha and Jignesh Patel for their "scribbles" on my NCR boards.

- Thanks to Mary and Kevin Beyer for their wonderful company in Madison and San Jose.

- Thanks to Abhinav Gupta, Chandra, and Sowmya for the "enlightening" visits to "Kapps" pizza in Mountain View.

- Thanks to Ravi Rao for our discusssions on medical database opportunities and later becoming a movie buddy.

It is fun hanging out with Emmanuel Ackaouy, Eric Rotenberg, Ajit Natarajan, Sriram Narasimhan, Ravi Murthy, Shivani Gupta, Ashutosh Dhodapkar, Sharmila Rajan, Archana Vasanthakumar, Vidhi, Chandru, Amit Paranjape, Ashutosh Singh, Vaishnavi Anjur, Sujan Chakraborthy, Lalita Subramanian, Avinash Sodani and Kanchen Rekhraj during my various years of graduate study.

I would like to thank my friends Venkat Chellasamy, Sivakumar Thoppe, Saravan Rajendran, Senthil Sankarappan, Ponnusamy Senthil Kumar, Mohammed Iqbal, Sam Rajarathnam, Murugavel Guruswami, Balasubramanian, Ravi Chandra, Chelian Pandian, Meenakshi Sundaram, Prakash Muthukrishnan, Prabhu and Ganapathy Subramanian. All of them wondered what was doing at school. Here it is folks!

Finally, I am grateful to my parents who stressed the importance of education and very supportive of me in all throughout my career. This thesis is dedicated to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The relational model was popularized in the 1970's and since that time has largely replaced the older hierarchical and network data models. Today, relational database systems are the dominant mainstream approach to database management and have a very healthy market share. Most recently, with the explosion of the web and e-commerce, the market for database systems has increased significantly as relational systems are being used as the backend for these systems. However, as new classes of applications began to be implemented over relational systems, several shortcomings of these relational systems were exposed. Object relational systems were developed to solve these problems by adding extensions to relational systems. These object relational systems can be characterized by starting with a relational system and adding:

- **Abstract Data Types:** The addition of abstract data types allows the creation and addition of new data types that a database system can understand. In an object relation system, these types are seamlessly treated as equivalent to built-in types. The definition of a new data type describes the data fields and the methods that operate on these fields. Method invocation can be specified as a part of the query. Adding a new data type requires the name of the type, storage information about the type, and routines to convert the data type from the internal representation to ASCII and back. The data types can be added either at compile-time or at run-time.

- **Type Constructors:** Type constructors are used in an object relational system to construct new types by composing base or abstract data types. The major class of type constructors included in these systems are composites (records), collections, and references. The class of collections can be further divided into sets, bags, arrays and lists.

- **Inheritance:** Inheritance allows the creation of new data types by derivation from existing types. For example, data inheritance allows new types to derive their data elements from their super types. On the other hand, function inheritance derives methods from the methods of super types. Furthermore, the inheritance mechanism typically allows the designer to refine the derived data elements and methods.

In this thesis our focus is on the set type constructor.

Set-valued attributes are beginning to appear in commercial O/R DBMSs — all the major vendor's universal servers either currently have set-valued attributes or plan to support them in the future. Furthermore, as set-valued attributes are part of the proposed SQL4 and OQL [Cat94] standards, it appears that in the future this support for set-valued attributes will not be optional — a system will provide set-valued attributes or it will not support the standard. Finally, the rise of XML as an important data standard increases the need for set-valued attributes, since it appears that set-valued attributes are key in the natural representation of XML data in relational systems [SHT$^+$99].

While the expressive power of set-valued attributes has long been studied by the data modeling and query language communities, to date there has been very little published about the efficient implementation of set-valued attributes in object relational systems. In this dissertation, we make a first step toward rectifying this lack of information by implementing and experimenting with several alternatives for set-valued attributes in an object-relational database system.

1.1 Set-Valued Attributes

To illustrate the usefulness of sets, consider the integration of mining in a database system that discovers association rules from market basket data. Suppose that the market basket data consists of sets of products (items) purchased together in a purchase transaction. An association rule expresses relationships of statistical significance between the presence of various items in these transactions. Association rules are of the form $X \Rightarrow Y$, where X (head) and Y (tail) are sets of items. Statistical significance is expressed in terms of support (fraction of transactions in which an item-set occurs) and confidence (fraction of transactions containing X also containing Y). The following relational schema could be used for storing the market basket data and rules.

1.1. Example.

Sales(Tid, Cid, Sid, Purchase-Date, Pid, Quantity, Price)

Customer(Cid, Name, Address)

Product(Pid, Name, Manufacturer, Release-Date, Category)

Store(Sid, Name, Address, Region)

Associations-Head(Rid, Pid)

Associations-Tail(Rid, Pid)

Now consider a query that mines the association rules on transactions that contain at least the products *swim-wear* and *goggles.* Such a query can be written using extended SQL as follows:

```
MINE RULES Associations-Head, Associations-Tail  AS
```

```
SELECT 1..n S1.Pid  AS BODY , 1..n S1.Pid  AS HEAD
FROM Sales S1
WHERE S1.Tid  IN ( SELECT S.Tid
                   FROM Sales S, Product P
                   WHERE S.Pid = P.Pid  AND P.Name = 'swim-wear'  OR
                         P.Name = 'goggles'
                   GROUP BY S.Tid
                   HAVING  COUNT (*) = 2 )
GROUP BY Tid
HAVING RULES WITH SUPPORT : 0.1, CONFIDENCE : 0.2                  (Q.1)
```

Such a query poses many problems in terms of expressability and performance. First, it is not very intuitive to express such a query in standard SQL. Second, the mining query requires a "groupby" operation on the transactions so that items belonging to each transaction can be operated upon simultaneously. Additionally, the query requires a join and a group by in order to locate the transactions containing both the specified items. Third, the correlation between the WHERE clause predicate and the HAVING clause predicates must be maintained when the query is changed. Finally, storing the rules (results of the query) requires multiple relations because of the first normal form limitation of the relational model.

Now consider the object relational version of the schema presented earlier, where the *Sales* relation contains a set-valued attribute describing the products bought by a customer in a transaction:

1.2. Example.

Sales(Tid, Cid, Sid, Purchase-Date, Items-Bought:{Pid, Quantity, Price})

Customer(Cid, Name, Address)

Product(Pid, Name, Manufacturer, Release-Date, Category)

Store(Sid, Name, Address, Region)

Associations(Head:{Pid}, Tail:{Pid})

Using this schema, Query (Q.1) can be rewritten in compact form as follows:

```
MINE RULES Associations  AS
SELECT 1···N S.Items-Bought.Pid  AS BODY , 1···N S.Items-Bought.Pid  AS HEAD
FROM Sales S
WHERE ( SELECT P.Pid
        FROM Product P
        WHERE P.Name = 'swim-wear'  OR P.Name = 'goggles' )
              SUBSET OF S.Items-Bought.Pid
HAVING RULES WITH SUPPORT : 0.1, CONFIDENCE : 0.2                    (Q.2)
```

Compared to the relational example, Query (Q.2) is more concise and intuitive. It does not incur the performance overhead of the extra groupby and join in query (Q.1), since all the products are already stored as a part of a single transaction tuple. Finally, only a single relation *Associations* is needed to store the rules.

1.2 Set-Valued Attribute Queries

In this section, we enumerate several types of queries involving set-valued attributes. In our examples we use the schema in Example 1.2 presented earlier. For the queries, let us assume that the *Pid* of the products *swim-wear* and *goggles* are *100* and *200* respectively.

- **Conjunctive Queries**: Conjunctive queries over set-valued attributes check for the existence of multiple elements in a set. An example that checks for a singleton member is:

```
SELECT S.Tid
FROM Sales S
WHERE 100  IN S.Items-Bought.Pid                                    (Q.3)
```

which retrieves all the transactions that contains the product *swim-wear*. Multiple memberships can be easily expressed as a conjunction of singleton membership as shown in the following query:

```
SELECT S.Tid
FROM Sales S
WHERE 100  IN S.Items-Bought.Pid  AND 200  IN S.Items-Bought.Pid    (Q.4)
```

The above query retrieves the transactions containing both *swim-wear* and *goggles*.

- **Disjunctive Queries**: Disjunctive queries contain predicates that are the disjunction of set element memberships. An example of such a query that presents the transactions containing either *swim-wear* or *goggles* is given below.

```
SELECT S.Tid
FROM Sales S
WHERE 100  IN S.Items-Bought.Pid  OR 200  IN S.Items-Bought.Pid     (Q.5)
```

- **Set Comparison Queries**: Set comparison queries are characterized by having predicates involving set comparisons, where one of the sets involved is explicitly enumerated or specified by a nested query. Some examples are shown below.

```
SELECT S.Tid
FROM Sales S
WHERE {100, 200} SUBSET OF S.Items-Bought.Pid                       (Q.6)
```

```
SELECT S.Tid
FROM Sales S
WHERE ( SELECT P.Pid
        FROM Product P
        WHERE P.Release-Date < S.Purchase-Date )
              SUBSET OF S.Items-Bought.Pid                    (Q.7)
```

Query (Q.6) finds all the transactions that contain both the products swim-wear and goggles and is equivalent to a conjunctive query. On the other hand, Query (Q.7) retrieves the transactions that contain all the products released before the date of purchase, which is not equivalent to a simple conjunctive query.

- **Nested Queries**: Since the elements of a set can be tuples, sets can be considered as relations, hence nested SQL queries can be posed. In these SQL queries, the nested queries can occur either in the projection or predicate lists, or in the FROM clause. They can potentially contain joins with other tables. We now enumerate a few examples:

```
SELECT S.Tid, ( SELECT  DISTINCT P.Manufacturer
                FROM Product P, S.Items-Bought B
                WHERE P.Pid = B.Pid )
FROM Sales S                                                  (Q.8)
```

Query (Q.8) lists all the sales transactions, and for each transaction lists the manufacturers of the products bought in that transaction. Next, as an example of a query in the predicate list, we consider a query that enumerates the transactions that contain at least one product bought on its release date:

```
SELECT S.Tid
```

```
FROM Sales S
WHERE  EXISTS ( SELECT P.Pid
                FROM Product P, S.Items-Bought B
                WHERE P.Pid = B.Pid  AND
                      S.Purchase-Date = P.Release-Date )                    (Q.9)
```

- **Set Containment Join Queries**: Set containment join queries involve joins in which the join predicate is set containment. Consider the following query

```
SELECT S1.Tid, S2.Tid
FROM Sales S1, Sales S2
WHERE S1.Items-Bought ⊆ S2.Items-Bought                                   (Q.10)
```

Query (Q.10) computes all pairs of transactions such that one transaction contains a subset of the items bought in the other transaction.

- **Set Intersection Join Queries**: Set intersection joins are closely related to set containment join queries. These queries involve joins in which the join predicate is intersection between set instances.

```
SELECT S1.Tid, S2.Tid
FROM Sales S1, Sales S2
WHERE S1.Items-Bought ∩ S2.Items-Bought ≠ φ                               (Q.11)
```

The above query computes pairs of sales transactions that have at least one item in common. Another variant of the query selects tuples such that the result of the intersection equals a given set.

```
SELECT S1.Tid, S2.Tid
```

```
FROM Sales S1, Sales S2
WHERE S1.Items-Bought ∩ S2.Items-Bought = {100, 200}                    (Q.12)
```

- **Set Predicate Join Queries**: We can generalize the containment and intersection joins to consider queries in which any set predicate is the join predicate. For example, consider:

  ```
  SELECT S1.Tid, S2.Tid
  FROM Sales S1, Sales S2
  WHERE FOR ALL x  IN S1.Items-Bought  AND
        FOR ALL y  IN S2.Items-Bought
        SUCH THAT x.Price > y.Price                                       (Q.13)
  ```

  This query computes pairs of transactions such that the price of each of the items from a tuple from S1 is greater than the price of all the items in transactions from a tuple in S2.

In this thesis we focus on the conjunctive and disjunctive set queries, and set containment join queries.

1.3 A Note on Modeling

Many real world concepts naturally lend themselves to be best described by sets. In such cases, set-valued attributes provide conciseness and ease of expression. There are two distinct possibilities for how sets can be used: as *logical collections* or as *nested relations.* Each of the usages can be best described by an example.

First, we consider sets as logical collections. Consider the following example that describes the set of movies seen by a person.

Moviegoer(name, street, city, state, zipcode, {movies})

On such a schema, we think most commonly occurring types of queries on sets will consider all elements together as a group. An example of such a query on the moviegoer relation includes finding all the moviegoers who have seen the movies *Titanic* and *As Good As It Gets*. These queries involve predicates containing well-defined set operations such as subset, superset, intersection and union. The sets proposed in GEM [Zan83] follow this general usage.

On the other hand, set-valued attributes can be considered equivalent to relations, as was done in the O2 [LRV88] object-oriented database system. In such a case, nested SQL queries can be posted over the sets. Consider again, the example schema quoted in Section 1.1. An example of a nested query is one that gets the set of manufacturers for the set of items sold in a sales record. Such a query requires examining the individual set elements rather than considering them as a group (in spite of their underlying grouping.)

In general we suspect that the nature of operations or predicates used in queries for one usage will differ from that in the other. However, nothing prevents the type of query that is natural in one usage from being posted on the other.

1.4 Contributions of the Thesis

This thesis proposes solutions to some of the problems described in the previous subsections. Specifically, the contributions of this research are as follows:

- **Set Storage Representations.** There are various alternatives for storing sets in a database. A straightforward approach stores the elements of set-valued attribute in one table in a different table, such that each element is a separate tuple. Such an approach is used in many commercial systems. However, it suffers from the fact that the set elements are scattered about the new table, and hence predicates that examine the set as a whole must perform groupings and may pose performance problems. Hence, we propose a new class of representations that group the set elements and store them together along with the rest of the attributes, thereby improving performance. In Chapter 2, we classify the space of representations and develop a detailed analytical model to evaluate the performance of various queries. These representations were implemented in an object relational system in order to explore the issues of ease of integration and also the performance implications and tradeoffs. The performance results and experience showed that nested representations perform better than the unnested representations for conjunctive and disjunctive queries. However, they have the drawback of extensive additions and modifications to the storage and query evaluation engine. Since the performance improvement is quite substantial, these modifications are justifiable.

- **Set Containment Joins.** The introduction of set-valued attributes gives rise to an interesting and challenging set of joins. One of the most obvious and challenging joins is the set containment join, which expresses complex queries in a concise way. Such a join selects pairs of tuples such that the set instance in one tuple is a subset of set instance in the other. Evaluating these joins is difficult, and naive approaches lead to algorithms that are expensive. In Chapter 4, we develop a new partition based algorithm called PSJ. This algorithm divides the input relations into partitions using a partitioning function on the set elements, and replicating some of the tuples as required. Each of the corresponding pairs partitions are individually joined using another level of partitioning based on set signatures. We compare the performance of PSJ with various algorithms by implementing in an object relational database system. The experiments show that PSJ outperforms other algorithms over a wide range of data distributions.

1.5 Outline of Dissertation

The remainder of this dissertation is divided into six chapters. Chapter 2 outlines options for storing sets. A performance evaluation of these storage representations is described in in Chapter 3. Chapter 4 defines the set containment join and proposes a new multi-level partition based algorithm called the Partitioned Set Join Algorithm. A detailed performance comparison of this algorithm with previously proposed algorithms — signature nested loops and the SQL approach — is presented in Chapter 5. Chapter 6 surveys the related work relevant to storage options and set containment joins. Finally, the conclusions and open issues for future work are presented in Chapter 7, which is followed by bibliography.

Chapter 2

Set Storage Representations

2.1 Introduction

While the expressive power of set-valued attributes has long been studied by the data modeling and query language design communities, to date there has been very little published about the options for implementing set-valued attributes and the performance implications of these options. In this chapter, we make a first step toward rectifying this lack of information by implementing and experimenting with several alternatives for set-valued attributes in an object-relational database system.

Two main options have been proposed for implementing set-valued attributes in object-relational systems: inlined, and external [Sto96]. In the inlined representation, the set itself is stored as a variable-length attribute within the tuple to which it belongs. On the other hand, in the external representation, the elements of a set-valued attribute are stored as tuples in an auxiliary table, and are connected back to the tuple to which they belong by a key-foreign key reference. While the O/R DBMS vendors generally do not publish their future design plans, it appears that at least initially O/R DBMS vendors will support sets using the external representation. The motivation for this is clear — the external representation has the overwhelming advantage of being simple to implement, since to the engine the set appears to be a standard table, and queries involving set-valued attributes are translated into standard relational operations on standard relational tables.

In our implementation we found that the external representation is indeed simple to implement, requiring no changes to the query evaluation engine. Unfortunately, the performance of the external representation on some common forms of queries involving set-valued attributes was abysmal when compared to the performance of our inline representation. Perhaps surprisingly, this was true even for large sets, which contradicts the common wisdom that only small sets should be inline [Sto96]. The performance advantage of inline sets was so dramatic that inline would always be the method of choice but for one major drawback: on queries that reference tables with set-valued attributes, but do not reference set-valued attributes, the overhead of inline sets can be an almost intolerable burden. Accordingly, we designed and implemented a third representation, nested external, which has the advantages of both.

In the nested external representation for sets, like the external representation of sets, the set elements are stored in an auxiliary table separate from the table with the set-valued attribute. However, in contrast to the external representation, in the nested external representation the set elements for a given set are gathered together in a single tuple. This tuple is "connected" back to the corresponding tuple in the table with the set-valued attribute by a key-foreign key relationship. Another way of putting this is that the nested external representation consists of vertically partitioning the inlined set representation form of the table with the set-valued attribute, with the set-valued attribute in one partition and the other attributes in the other partition. Although the nested external representation requires more changes to the query evaluation engine than the external representation (including new predicate evaluation primitives, and nest and unnest operators), our experiments show that the performance gain from nested external makes these changes well worth the trouble.

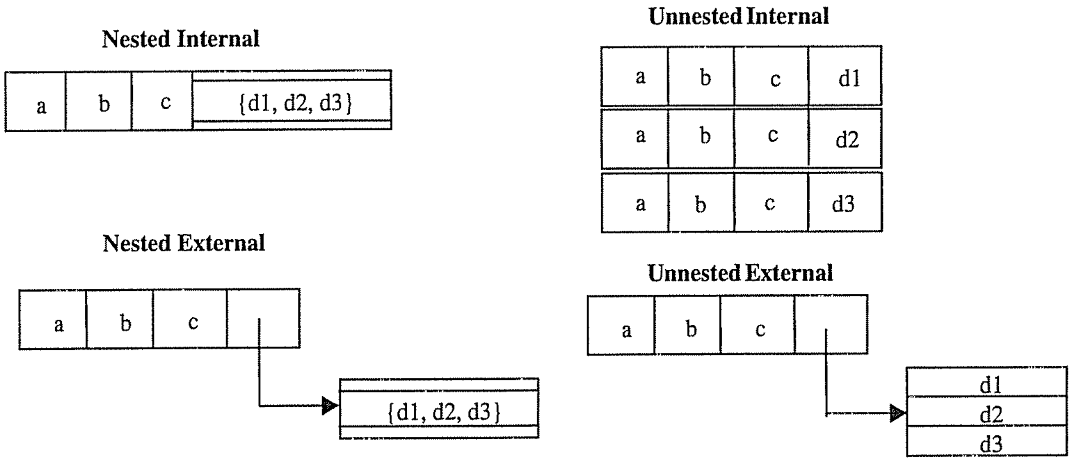


Figure 1: Storage Options for Set Valued Attributes

2.1.1 Example

Let us consider a relation $R(a, b, c, \{d\})$. As we have mentioned, there are various options for storing set-valued attributes: store along with the rest of attributes in the relation (nested internal), vertically decomposed and stored in a separate relation (nested external), each element of the set is stored as a tuple in a separate relation (unnested external) or replicate rest of the attributes for each element in the set (unnested internal). These storage options are enumerated in Figure 1.

While set-valued attributes have been considered in the past, our work differs from the existing body of work in many aspects. We specifically focus on the issues related to storing set-valued attributes. We classify the space of representations based on certain characteristics and analyze each of them in detail using an analytical model. The nested internal representation is similar to Normalized Storage Model (NSM) [HO88] in terms of storage structure. On the other hand, the nested external representation is a variation of the Decomposed Storage Model(DSM) [CK85] where only the instances of set-valued attributes are stored in a separate relation. Neither model has been considered specifically in terms of supporting set-valued attributes. In our work we focus on set-valued attributes, and quantify the performance of these representations under various queries on an operational object-relational system.

2.1.2 Chapter Organization

This chapter is organized as follows. The individual representations are described in Section 2.3. The general analytical framework is described in Section 2.4. A description of the implementation of the nested internal representation and evaluation of queries over the representation are presented in Section 2.5. Similarly, Section 2.6 describes the unnested external representation and queries over that representation, while Section 2.7 covers the nested external representation. Finally, Section 2.8 concludes by summarizing the chapter.

2.2 Desirable Characteristics of a Set Representation

A set representation provides storage for the set-valued attributes of a relation. In addition to providing storage, it requires other properties to be practical. In this section, we identify the properties an ideal representation should have.

- **Composability**: From the data modeling perspective, "set" is considered to be a type constructor, and hence it should be composable with any arbitrary type. The composability property requires that such orthogonality also be maintained at the storage level. To be more concrete, consider object relational systems that provide type extensibility by allowing the user to declare and define their own types in the form of ADTs. Each ADT has its own database and memory representation that is optimized for storage and computation. Also, the size of these representations can vary widely. A set representation should honor the properties of these ADTs and seamlessly convert the ADTs into their required representation when required. It should provide uniform performance irrespective of the size of its element, and further provide evaluation mechanisms to invoke methods on the element ADTs.

User defined ADTs might need support for aggregate properties when they are contained in a set. These aggregate properties can be either user-defined as a part of the ADT definition, or system defined. An aggregate property is a characteristic of the entire set. For example, an aggregate property of a set of polygon ADTs might be a bounding box encompassing all the polygons. An example of a system defined aggregate property is the cardinality of set. A set representation should efficiently support storage and retrieval of these properties.

- **Handling Varying Set Cardinality and Sizes**: A set can vary in size as new elements are added and existing elements are deleted. A single set representation should uniformly handle variations in the number of elements as well as the size of the elements. It should lend itself for queries to be evaluated efficiently irrespective of whether the sets are considered as logical collections or as relations.

- **Nesting**: Since set is a type constructor, and set of an arbitrary type is a new type, one can declare a set-valued attribute that is a set of sets. Such a recursive definition can lead to an arbitrary depth of nesting. A single set representation should be able to store such recursive structures and also provide reasonable performance when queries are posed.

- **Updateability**: A set representation should provide capabilities for adding, deleting and modifying its elements. In an ideal representation, the effect of these modifications should be localized. In other words, modifying an element should not affect other elements or other attributes. In addition to individual element updates, modification should allow efficient addition and deletion of the entire set. Such operations are required when an entire base tuple is added or deleted.

- **Ease of Integration**: Most of the commercial vendors of relational systems are now moving to the object relational market. As Stonebraker [Sto96] puts it, vendors have multiple options for building object relational products. The option of rewriting a relational engine from scratch requires a lot of effort, in terms of investment, and may have a negative impact on timely delivery as well as reliability. Hence relational vendors have resorted to incremental evolution of their existing systems. Any object relational feature not only should easily integrate with existing code with minimal modifications but should have the ability to meet the performance requirements set forth by standard benchmarks. Any set representation added to the system should have this feature.

Addressing all of these concerns is a large task. In this thesis we focus on the efficient support of operations over set-valued attributes.

2.3 Taxonomy of Set Organizations

The efficient evaluation of queries involving set-valued attributes depends on how these attributes are stored in the database and the characteristics of the predicates evaluated against them. In this section, we enumerate the feasible representations and classify them depending on their characteristics, as shown in Figure 2.

2.3.1 Homogeneous Vs Heterogeneous

Set representations can be broadly classified as *homogeneous* or *heterogeneous.* Systems with homogenous representations use the same representation to store all the set instances in a relation, whereas systems with heterogenous systems may use different representations for the sets in different tuples of the relation. Homogenous representations provide a uniform view to the system and hence are amenable for representation specific optimizations that an optimizer can take advantage of during query transformation or query rewrite. However, the representation might not be the right choice for some instances of sets, for example, when some instances are much larger than others.

In heterogeneous representations, one could consider schemes in which the tuples of the table each use the representation that is most appropriate for their instance of the set-valued attribute depending upon its size, access frequencies and other characteristics. However, adopting such a heterogeneous scheme significantly increases the complexity of query optimization and evaluation. Query optimization becomes hard since each representation generates different plans, thereby exploding the search space, while query evaluation becomes hard because the query evaluation engine must now be prepared to execute different plans on a per-tuple basis. For this reason we have not considered heterogeneous schemes in our research, although they represent an interesting topic for future work.

2.3.2 Nesting Vs Location

Set storage representations can be further classified based on two orthogonal characteristics: *nesting* and *location.* The characteristic of nesting describes whether the elements in the set are grouped together or scattered. On the other hand, location specifies whether

	Internal	External
Unnested	No	Yes
Nested	Yes	Yes

Figure 2: Taxonomy of Set Representations

the set elements are stored along with the rest of the attributes in the relation or vertically decomposed and stored separately. Hence, the four feasible representations are *Nested Internal*, *Nested External*, *Unnested Internal* and *Unnested External.* This classification is shown in Figure 2. We concentrate on all the representations with the exception of Unnested Internal. There are two reasons for not considering this representation. First, unnested internal replicates the rest of attributes for each set element, thereby consuming a large amount of storage. Second, this replication leads to update anomalies.

2.3.3 Indexed Representations

All the aforementioned representations can be augmented with indices to speed up the evaluation of predicates. Specifically, the indices could potentially improve the performance of membership testing and gathering the elements of set. Again, the indices for sets can be either nested or unnested. Nested indices treat each set as a single entity, whereas unnested indices treat each set element as an indexable entity. Some examples of nested indices include signature files and RD-Trees, whereas B-Trees can serve as unnested indices. We restrict ourselves to unnested indices because unlike the other indices, all relational systems support some index like a B-Tree, and one focus of our work is to try to answer the basic question of how effective an existing relational system can be in storing sets.

For the nested internal representation, the index on the set-valued attribute decomposes the sets to individual elements and maps a given set element to the set of tuple identifiers where that element occurs. Depending on the type of representations, two different types of indices can be created. Since external representations store sets in a different relation, we can create a single relation index or a join index [Val87] connecting the base relation and the set relation.

2.4 Analytical Model

To predict the cost of query evaluation and the space requirements of the representations, we develop a detailed analytical model. The purpose of this model is not to predict absolute performance, but rather to identify the trends and characteristics in the relative costs of the representations. We consider each representation in turn and derive equations describing the cost of evaluation of queries and the overall space requirements for each representation. The query evaluation cost is furthermore broken down into I/O cost and CPU cost. To describe the model, we consider a relation $R(a, b, c, \{d\})$, which contains the set-valued attribute d. The model takes as input, different parameters for different representations. Table 1 describes the analytical parameters used in all representations.

The queries we chose represent the main classes of set-valued queries: conjunctive queries and disjunctive queries. For external representations, to simplify the analytical modeling, we assume that the base relation fits in memory. The queries we consider are defined using the schema $R(a, b, c, \{d\})$. The following queries are evaluated using the

Notation	Description
$\mid R \mid$	Number of tuples in relation R
$\mid\mid R \mid\mid$	Number of pages in relation R
k_R	Average number of elements per set (cardinality) in relation R
A_s	Average size of each set element in relation R
A_{abc}	Average size of the attributes a, b and c in relation R
S_s	System space overhead for storing a tuple
K_s	Space required for a key for external representations
B_t	CPU cost of accessing a tuple from the buffer pool
TID_s	Size of the tuple identifier
PID_s	Size of page identifier
IO_{seq}	Cost of a sequential I/O
IO_{rand}	Cost of a random I/O
P_s	Page size
σ	Selectivity of a predicate on set
N	Number of elements in the predicate
TA_t	Tuple assembly cost if the tuple is scattered across multiple pages
hi_t	Cost of inserting an element into hash table
hc_t	Cost of computing the hash function on an element
hp_t	Cost of probing an element in the hash table
cp_t	Cost of comparing two set elements
M	Amount of memory available in the system

Table 1: Description of Notation Used

analytical model.

- **Conjunctive Queries:** In this class, we ran two queries: one in which the set-valued attribute does not appear in the result and the other in which it appears in the result.

  SELECT $R.a$, $R.b$, $R.c$

  FROM R

  WHERE (*"pred-elem-1"*, ... *"pred-elem-N"*) SUBSET OF $R.d$ $(Q.14)$

  SELECT $R.a$, $R.b$, $R.c$, $R.d$

  FROM R

  WHERE (*"pred-elem-1"*, ... *"pred-elem-N"*) SUBSET OF $R.d$ $(Q.15)$

- **Disjunctive Queries:** Again in this class, we ran two queries: one in which the set-valued attribute does not appear in the result and the other in which it appears in the result.

```
SELECT R.a, R.b, R.c
FROM R
WHERE "pred-elem-1" IN R.d OR ... OR "pred-elem-N" IN R.d                (Q.16)
```

```
SELECT R.a, R.b, R.c, R.d
FROM R
WHERE "pred-elem-1" IN R.d OR ... OR "pred-elem-N" IN R.d                (Q.17)
```

2.5 Nested Internal Representation

In this representation, the set elements are grouped together and stored along with the rest of the attributes. Typically, in an RDBMS, a relation is stored as a set of tuples in a file that contains a collection of slotted pages. Each tuple contains instances of all attributes in the relation as specified in the schema. Since a set-valued attribute is of variable length, it can be either expanded in place or stored "at the end" of tuple. With multiple variable length attributes things are still more complex, since they cannot all be "at the end" of the tuple. The standard solution in this case is to store variable length attributes in a heap area at the end of the tuple, with offsets into this heap stored in the body of the tuple. This allows one to extract any variable length attribute of the tuple without scanning other variable length attributes. We have adopted this approach for the internal set-valued attributes; an instance of the internal representation in a tuple is shown in Figure 3.

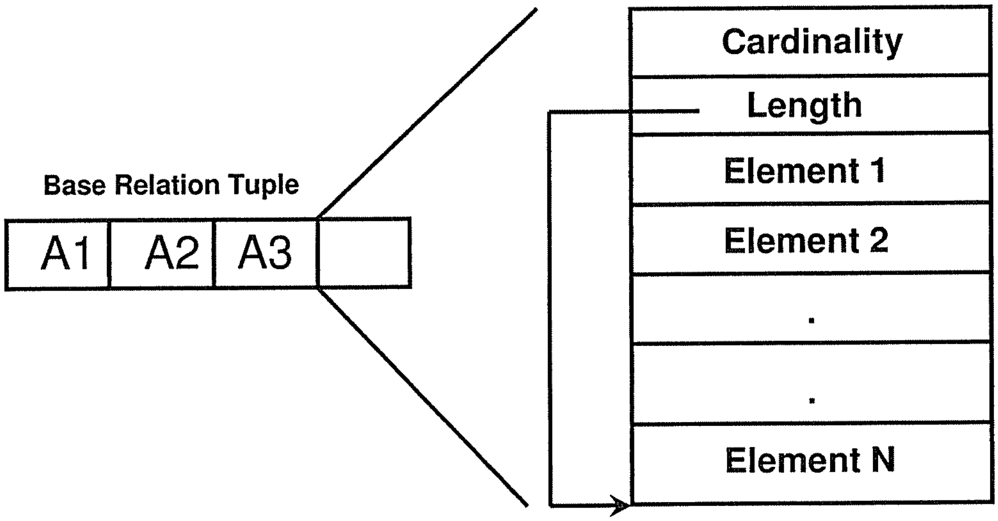


Figure 3: Nested Internal Representation of a Set in a Tuple

The internal representation of a set consists of a header and a body. The header describes the meta information of the set: its cardinality and its length in bytes. The reason for storing the cardinality is that it serves as a signature for subset, superset and equality operations on sets, and it marks the end of the set during iterator operation. For the indexed nested internal representation, the set-valued attribute is unnested and an unclustered index is created that maps the individual set elements to a collection of tuple identifiers.

2.5.1 Handling Large Sets

Since sets are generic collections, there is no upper bound on the number of elements a set can hold, so the internal representation can potentially become large and a tuple containing such a set might overflow to multiple pages. Hence, a mechanism is required to handle large tuple sizes. There are various well-known approaches to handle large tuples. A simple and straightforward approach is to move the tuple to a list of chained pages and replace the tuple in the original page with a forward record pointing to the first page in the chain. Another approach includes a directory structure such as B-Tree on the multiple data pages of the tuple. The root page identifier of the directory structure is stored in the forward record. The directory structure maps the start offset of the page relative to the tuple to the page identifier. We chose the later approach since it is already available in the system and also our work does not focus on the tradeoffs of various approaches for large tuples. Such large tuples incur the additional cost of multiple disk seeks and assembling the tuple together contiguously in memory before submitted to an operator. If the sets are too large too fit in memory, then individual fragments are brought into memory depending on the access pattern.

2.5.2 Evaluation of Set Predicates

A set predicate expresses a boolean condition over the domain of sets. A set instance should satisfy the condition in order to qualify. Set predicates are expressed using set operations described in Chapter 1. Since the nested internal representation groups the set elements together, we can take advantage of this property during the evaluation of set predicates. A naive approach for evaluation builds a hash table for the entire set, with predicate elements probed into this hash table. The problem with this approach is that individual sets might be larger than memory, which requires specialized operators to be written. In the refined approach, the predicate elements are staged in a hash table and the individual set elements are probed. This approach has the advantage of not bringing the entire set in memory during evaluation and also enables premature termination of the scan of the set once it can be determined whether predicate has been satisfied or will not be satisfied (e.g., with "short-circuit" evaluation of AND and OR operations), thereby saving I/O cost. Both the approaches have a linear predicate evaluation cost proportional, to the number of elements in the set, which is $O(n)$.

2.5.3 Analytical Evaluation

In this section, we compute the space requirements and the cost of query evaluation for nested internal and indexed nested internal representations.

Case 1: Nested Internal Representation

The total space requirements for the nested internal representation NI_{SPACE} is given by

$$NI_{SPACE} = \| R \| + \left\lceil \frac{S_s \mid R \mid}{P_s} \right\rceil \tag{1}$$

assuming that the tuples are tightly packed. However, in a real system, there would be fragmentation causing the space requirement to go up slightly.

Query evaluation using the nested internal representation is straightforward. The set predicate is applied to each tuple in relation R, and the qualified tuples are stored in the result relation. The I/O cost $NICQ_{IO}$, and CPU cost $NICQ_{CPU}$ for conjunctive query with no set in the result is given by

$$NICQ_{IO} = \begin{cases} \left(\| R \| + \left\lceil \frac{S_s|R|}{P_s} \right\rceil\right) IO_{seq} + & \text{I/O cost of reading relation } R \\ \left\lceil \frac{\sigma|R|(A_{abc}+S_s)}{P_s} \right\rceil IO_{seq} + & \text{I/O cost of writing the result tuples} \\ TA_t \times \mid R \mid + & \text{Tuple assembly cost for large tuples} \\ (1+\sigma) \mid R \mid B_t & \text{Cost of accessing tuples from buffer pool} \end{cases} \tag{2}$$

$$NICQ_{CPU} = \begin{cases} (hc_t + hi_t)\, N + & \text{Hash table construction for predicate elements} \\ (hc_t + hp_t) * k_R \mid R \mid & \text{Probing hash table with set elements} \end{cases} \tag{3}$$

The CPU cost represents the worst case cost since it assumes that all the set elements are probed into the predicate hash table. The cost of evaluating a conjunctive query with the set in the result is similar except that the cost of writing the result tuples becomes higher. Hence the I/O cost $NICQWS_{IO}$ is given by

$$NICQWS_{IO} = \left\{ \begin{array}{ll} \left(\| R \| + \left\lceil \frac{S_s|R|}{P_s} \right\rceil \right) IO_{seq} + & \text{I/O cost of reading relation } R \\ \left\lceil \frac{\sigma|R|(A_{abc}+k_R A_s+S_s)}{P_s} \right\rceil IO_{seq} + & \text{I/O cost of writing the result tuples} \\ TA_t \times \mid R \mid + & \text{Tuple assembly cost for large tuples} \\ (1+\sigma) \mid R \mid B_t & \text{Cost of accessing tuples from buffer pool} \end{array} \right. \tag{4}$$

The CPU cost is the same as $NICQ_{CPU}$. In both the cases, we assume that enough memory is available to hold the result so that it is written in sequential fashion. The cost of evaluating disjunctive queries with and without the set in the result is identical to the case for conjunctive queries.

Case 2: Indexed Nested Internal Representation

For the indexed nested internal representation, determining space requirements requires estimating the number of index pages. Let

$$K = \left\lceil \frac{P_s}{A_s + PID_s + S_s} \right\rceil \quad \text{and} \quad L = \left\lceil \frac{P_s}{A_s + TID_s + S_s} \right\rceil \tag{5}$$

be the number of entries that fit on a non-leaf page and leaf page respectively. Then the total number of leaf level pages in the index LP_{SPACE} is given by

$$LP_{SPACE} = \left\lceil \frac{k_R \mid R \mid}{L} \right\rceil \tag{6}$$

Consequently, the total space requirements for indexed nested representation INI_{SPACE} is the sum of space for the relation and the space for the index.

$$INI_{SPACE} = \begin{cases} \left(\| R \| + \left\lceil \frac{S_s|R|}{P_s} \right\rceil \right) IO_{seq} + & \text{Total number of pages the relation spans} \\ \frac{2K \times LP_{SPACE}}{K-1} & \text{Total size of the index} \end{cases} \tag{7}$$

assuming that the key is duplicated, since the index is non-unique.

Query evaluation for the indexed nested internal representation requires probing the index with predicate elements. The TIDs are either ANDed or merged depending on the whether the query is conjunctive or disjunctive. The result TIDs are sorted to prevent random disk seeks when the tuples are actually fetched. The I/O cost $INICQ_{IO}$ is given by

$$INICQ_{IO} = \begin{cases} N log_K \left(LP_{SPACE} \right) IO_{rand} + & \text{Probing predicate elements in the index} \\ \sigma \mid R \mid IO_{rand} + & \text{Cost of fetching the tuples} \\ \sigma \left\lceil \frac{(A_{abc}+S_s)|R|}{P_S} \right\rceil IO_{seq} + & \text{Cost of writing the result} \\ 2\sigma \mid R \mid B_t & \text{Fetching tuples from buffer pool} \end{cases} \tag{8}$$

assuming that each result tuple is in a separate page. The CPU cost $INICQ_{CPU}$ is given by

$$INICQ_{CPU} = \begin{cases} N log_2 \left(K \right) log_K (LP_{SPACE}) cp_t + & \text{Searching index pages using binary search} \\ N\sigma \mid R \mid cp_t & \text{ANDing or merging TIDs} \end{cases} \tag{9}$$

assuming that binary search is used in the index page to locate an entry. As stated earlier, the cost of evaluating a conjunctive query with set in the result is similar except for the added cost of writing the result as in $NICQWS_{IO}$.

2.6 Unnested External Representation

In external representations, the relation containing the set-valued attribute is decomposed into multiple relations. If the relation has only a single set-valued attribute, then there are just two relations, one for the "base tuple," the other for the elements of the sets. For simplicity we will assume that there is only a single set-valued attribute. We explore two variants of the external representation: *External* and *Nested External.* Because of the decomposition between a tuple and its set-valued attribute, we need a mechanism to compose the base tuple and its set elements; we use a key-foreign key relationship for this purpose.

In this representation, the set-valued attributes are stored separately from the base relation. All the sets are unnested and each element is stored as a separate tuple. Consider a relation $R(a, b, c, \{d\})$. This relation is decomposed into two relations: $R_B(i, a, b, c)$ and $R_S(i, d)$, where R_S contains all the set elements. The set elements and the base tuple are related by the key attribute i. During query processing, an equijoin is performed to associate the base tuple and the set-elements.

2.6.1 User Defined Keys Vs System Defined Keys

The key attribute can be either user-defined or system generated. If a key is being declared as a part of the base relation definition, the same set of attributes can be used as the key. In other cases, the system should supply the key. There are pros and cons of user defined and system generated keys. User defined keys can be composite and become potentially large in proportion to the average size of the set element. This would result in a dramatic increase of I/O cost. User defined keys are also exposed to the possibility of schema modification for keys, which will lead to the reorganization of the base and set relations.

System generated keys are small and immune to modification by the user. The *next key* to be used can be stored in the catalog entry for the base relation. When a new tuple is inserted, this next key is used and updated. However, this approach presents problems when multiple tuples are being inserted in a single transaction. The catalog has to be locked for the duration of the transaction, which in turn serializes all the transactions thereby reducing the concurrency in the system. Possible alternatives that alleviate these problems are the use of either the *physical rid of the base tuple* or the *time of insertion* as the key. Physical rids can be as long as 32 bytes, which might be unsuitable for smaller types (integer, float etc.) since the space overhead is 2 to 4 times the size of the type. The time for insertion also suffers from the same problem. Such an increase in space also decreases the performance of some types of queries, due to increased I/O cost.

2.6.2 Clustering

A relation R_S can be stored in such a way that the set elements belonging a given tuple in $R_B(i, a, b, c)$ are clustered together. Such an organization has the advantage of fast retrieval of set elements, especially for queries that output the set in the result. However, such an organization is not useful unless clustered indices are created on the key attribute i in R_S. The indexed variant of the unnested external representation creates two indices on relation R_S: a clustered index on the key attribute i to facilitate faster grouping of elements in a set, and an unclustered index on set elements to speed up predicate evaluation on set elements.

2.6.3 Query Rewriting

In this section, we describe how conjunctive and disjunctive queries can be rewritten for unnested external representation. There are two approaches for rewriting the queries: *disjunctive groupby* and *cascaded joins.*

Disjunctive Groupby

In this approach, we exploit the semantic property that individual elements in a set instance are distinct. Now the conjunctive query (Q.14) can be rewritten as

SELECT $B.i$, $B.a$, $B.b$, $B.c$

FROM R_B B, R_S S

WHERE $B.i = S.i$ AND ($S.d =$ *"pred-elem-1"* OR $\ldots S.d =$ *"pred-elem-N'*)

GROUP BY $B.i$, $B.a$, $B.b$, $B.c$

HAVING COUNT (*) $= N$ $\qquad (Q.18)$

This approach has the deficiency that relational systems tend to choose plans involving sequential scans since it is difficult to estimate the selectivity of a disjunctive predicate. Hence, this approach might be suited when there are no indices or the selectivity is low, which precludes the use of indices. Also, the extra CPU cost of group by $B.i$, $B.a$, $B.b$, and $B.c$ is incurred even though the grouping is required only on $B.i$.

Conjunctive queries with sets in the result (Q.15) are transformed into two queries: one for determining the membership, and the other for grouping all the elements of the set. The intermediate result of membership is stored in a temporary table before it is fed for grouping the elements in a set. Since the membership query filters the set elements not satisfying the predicate, we require the second query to group these remaining elements of the set. The queries are shown below.

INSERT INTO $Temp(i, a, b, c)$

SELECT $B.i$, $B.a$, $B.b$, $B.c$

FROM R_B B, R_S S

WHERE $B.i = S.i$ AND ($S.d =$ "*pred-elem-1*" OR ... OR $S.d =$ "*pred-elem-N*")

GROUP BY $B.i$, $B.a$, $B.b$, $B.c$

HAVING COUNT (*) $= N$ $(Q.19)$

SELECT $T.a$, $T.b$, $T.c$, $S.d$

FROM *Temp* T, R_S S

WHERE $T.i = S.i$ $(Q.20)$

The second query shows that the results are not grouped and the attributes a, b and c are repeated for every set element. This illustrates the major drawback of "lack of grouping" in the unnested external representation. Because of the lack of grouping, tuple blowup occurs, which causes an increase in storage as well as in the processing cost up in the query tree.

Cascaded Joins

In this approach, as outlined in [CDN$^+$97], the base relation is involved with multiple joins on the set relation each relating a predicate element with the base relation. Hence the number of joins is proportional to the number of predicate elements. The rewritten query for the conjunctive query (Q.14) is shown below:

SELECT $B.i$, $B.a$, $B.b$, $B.c$

FROM R_B B, R_S S_1, R_S S_2 ..., R_S S_N

WHERE $B.i = S_1.i$ AND $B.i = S_2.i$ ... AND $B.i = S_N.i$ AND

$S_1.d =$ "*pred-elem-1*" AND ... AND $S_N.d =$ "*pred-elem-N*" $(Q.21)$

Such a query rewrite might perform well in the presence of indices for limited selectivities.

For queries that project the set in the result, cascaded joins require an extra join with R_S, as shown below:

SELECT $B.i$, $B.a$, $B.b$, $B.c$, $S.d$
FROM R_B B, R_S S_1, ..., R_S S_N, R_S S
WHERE $B.i = S.i$ AND $B.i = S_1.i$, ..., AND $B.i = S_N.i$ AND
$S_1.d =$ *"pred-elem-1"* AND ... AND $S_N.d =$ *"pred-elem-N"* $(Q.22)$

Disjunctive queries without sets can be rewritten easily as a join with a disjunctive predicate, as shown below. For the set to be in the result, the projection list should be augmented with $S.d$ and an extra join is required to assemble all the set elements.

SELECT $B.i$, $B.a$, $B.b$, $B.c$
FROM R_B B, R_S S
WHERE $B.i = S.i$ AND ($S.d =$ *"pred-elem-1"* OR ... OR $S.d =$ *"pred-elem-N"*) $(Q.23)$

2.6.4 Analytical Evaluation

In this, we derive the space requirements and the cost of query evaluation using the analytical model for unnested external and indexed unnested external representations.

Case 1: Unnested External Representation

The space requirements for the decomposed relations R_B and R_S is given by

$$\| R_B \| = \left\lceil \frac{(A_{abc} + K_s + S_s) \mid R \mid}{P_S} \right\rceil \quad \text{and} \quad \| R_S \| = \left\lceil \frac{(A_s + TID_s + S_s) k_R \mid R \mid}{P_S} \right\rceil \tag{10}$$

Hence the total space requirements for unnested external representation UNE_{SPACE} is given by

$$UNE_{SPACE} = \| R_B \| + \| R_S \| \tag{11}$$

As evident from the equations, the total space increases because of the extra space required for key and system overhead for each set element being stored as a tuple.

As stated earlier, we assume that the relation R_B fits in memory. However, after the join, the number of tuples could potentially blowup, which will require the partial intermediate relation being saved to disk and read back. We further assume that a hash join is used. For conjunctive queries with no set in the result we choose to rewrite using the disjunctive groupby approach, so the I/O cost $UNECQDG_{IO}$ and CPU cost $UNECQDG_{CPU}$ are given by

$$UNECQDG_{IO} = \begin{cases} (\| R_B \| + \| R_S \|)IO_{seq} + & \text{I/O cost of reading the base and set relation} \\ (1 + k_R) \mid R \mid B_t & \text{Buffer pool cost for reading} \\ 2\left(\left\lceil \frac{(A_s+A_{abc}+S_s+K_s)\sigma|R|}{P_s} \right\rceil - M\right) IO_{seq} + & \text{Reading and writing the partial temp relation} \\ 2\sigma \mid R \mid B_t + & \text{Buffer pool cost for reading and writing temp relation} \\ \left\lceil \frac{(A_s+A_{abc}+S_s+K_s)\sigma|R|}{P_s} \right\rceil IO_{seq} + & \text{I/O cost of writing the result} \\ \sigma \mid R \mid B_t & \text{Buffer pool cost for writing result} \end{cases} \tag{12}$$

$$UNECQDG_{CPU} = \left\{ \begin{array}{ll} \frac{k_R N||R||cp_t}{2} + & \text{Average cost of evaluating disjunctive predicate} \\ (hc_t + hi_t) \mid R \mid + & \text{Constructing hash table for } R_B \text{ during join} \\ (hc_t + hp_t)\sigma \mid R \mid + & \text{Probing join hash table with qualified tuples} \\ 4hc_t\sigma \mid R \mid + & \text{Computing the hash function for 4 group by attributes} \\ hp_t\sigma \mid R \mid & \text{Probing the hash table for group by} \end{array} \right. \tag{13}$$

This I/O cost assumes that the result of join does not fit in memory, since a blow up could occur by a factor of R_B proportional to number of elements in the predicate, especially when the selectivity is low.

Case 2: Indexed Unnested External Representation

Total space requirements for the indexed unnested external representation $IUNE_{SPACE}$ consists of the size of the decomposed relations, the size of the clustered index on the key attribute i, and the size of an unclustered index on the attribute d on relation R_S. The number of keys that fit in a leaf page and an intermediate page are given by

$$KP = \left\lceil \frac{P_s}{K_s + PID_s + S_s} \right\rceil \quad \text{and} \quad LP = \left\lceil \frac{P_s}{K_s + TID_s + S_s} \right\rceil \tag{14}$$

Hence the total number of leaf level pages in the clustered index LK_{SPACE} is given by

$$LK_{SPACE} = \left\lceil \frac{k_R \mid R \mid}{LP} \right\rceil \tag{15}$$

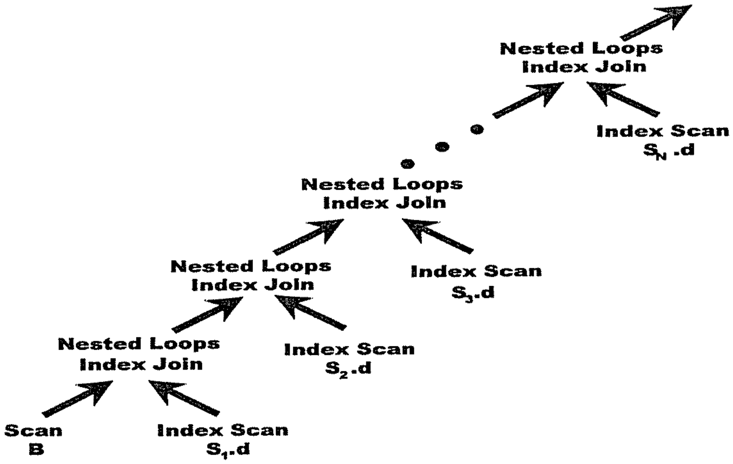


Figure 4: Execution Plan for Cascaded Joins

and the total space $IUNE_{SPACE}$ is

$$IUNE_{SPACE} = \begin{cases} \| R_B \| + \| R_S \| + & \text{Size of the relations} \\ \frac{2K \times LP_{SPACE}}{K-1} + & \text{Size of the unclustered index on } d \\ \frac{2KP \times LK_{SPACE}}{KP-1} & \text{Size of the clustered index on } i \end{cases} \tag{16}$$

For query rewrites using cascaded joins, we use the plans of the type shown in Figure 4 to simplify modeling. Many plans can be generated from this template depending on the placement of the join with R_B. The placement strictly depends on the selectivity of the individual set elements. The I/O cost $UNECQCJ_{IO}$ and CPU cost $UNECQCJ_{CPU}$ are given by

$$UNECQCJ_{IO} = \begin{cases} \| R_B \| IO_{seq}+ & \text{I/O cost of scanning relation } R_S \\ log_K(LP_{SPACE})IO_{rand}+ & \text{Probing unclustered index scan on } R_S.d \\ \sigma \mid R \mid IO_{rand}+ & \text{Fetching qualified tuples (from unclustered index)} \\ (N-1)log_{KP}(LK_{SPACE})IO_{rand}+ & \text{Probing clustered index on } R_S.i \\ (N-1)\sigma \| R_S \| IO_{seq}+ & \text{Fetching qualified tuples (from clustered index)} \\ (N+1)\sigma \mid R \mid B_t+ & \text{Fetching qualified tuples from buffer pool} \\ \left\lceil \frac{(A_s+A_{abc}+S_s+K_s)\sigma|R|}{P_s} \right\rceil IO_{seq} & \text{I/O cost of writing the result} \end{cases} \quad (17)$$

$$UNECQCJ_{CPU} = \begin{cases} log_2(K)log_K(LP_{SPACE})cp_t+ & \text{Searching unclustered index pages - binary search} \\ (N-1)log_2(KP)log_{KP}(LK_{SPACE})cp_t+ & \text{Cost of searching clustered index} \\ \sigma \mid R \mid^2 cp_t & \text{Cost of comparing with tuples in } R_B \end{cases} \quad (18)$$

2.7 Nested External Representation

Nested external is similar to nested internal except that set-valued attribute is vertically decomposed and stored in an external relation. Consider a relation $R(a, b, c, \{d\})$. This relation is decomposed into two relations: $R_B(i, a, b, c)$ and $R_S(i, \{d\})$, where R_S contains all the set elements. The elements and the base tuple are related by the key attribute i. The number of tuples in R_B and R_S is the same as in un-decomposed relation R. During query processing, an equijoin is performed to associate the base tuple and set-elements. For the indexed nested external representation, the set valued attribute in R_S is unnested and an unclustered index is created that maps the individual set elements to a collection of tuple identifiers in R_S. Most of the issues related with nesting discussed in Section 2.5 are applicable to nested external. Since the set-valued attribute is stored separately from the base relation, issues about user defined versus system defined keys are relevant in this representation.

Query rewriting is relatively easy as compared to the unnested external representation. Queries are rewritten such that the set predicates are evaluated as in the nested internal representation and satisfying tuples in R_S are related with the relation R_B using a join. For the indexed variant, the index is probed for each set element in the predicate and TIDs are either ANDed or merged (depending on whether it is a conjunctive or disjunctive query) and tuples are fetched before fed into the join.

2.7.1 Analytical Evaluation

In this section, we compute the space requirements and the cost of query evaluation for nested internal and indexed nested internal representations.

Case 1: Nested External Representation

The space requirements for the vertically decomposed relations R_B and R_S are given by

$$\| R_B \| = \left\lceil \frac{(A_{abc} + S_s + K_s) \mid R \mid}{P_s} \right\rceil \quad \text{and} \quad \| R_S \| = \left\lceil \frac{(k_R A_s + S_s + K_s) \mid R \mid}{P_s} \right\rceil \tag{19}$$

Total space requirements for the nested external representation NE_{SPACE} is given by

$$NE_{SPACE} = \| R_B \| + \| R_S \| \tag{20}$$

We simplify the analytical modeling for queries, by assuming that the relation R_B fits in memory. We assume that a hash join is used. Query evaluation cost for conjunctive queries with no set in the result is given by

$$NECQ_{IO} = \begin{cases} \| R_B \| IO_{seq} + & \text{I/O cost of reading the base relation for joining} \\ \| R_S \| IO_{seq} + & \text{I/O cost of reading the set relation} \\ TA_t \mid R \mid + & \text{Tuple assembly cost} \\ (2+\sigma) \mid R \mid B_t + & \text{Buffer pool cost reading and writing the result} \\ \left\lceil \frac{\sigma |R| (A_{abc} + S_s) IO_{seq}}{P_s} \right\rceil & \text{writing the result tuples} \end{cases} \tag{21}$$

$$NECQ_{CPU} = \begin{cases} (hc_t + hi_t) N + & \text{Constructing hash table for predicate elements} \\ (hc_t + hp_t) \mid R \mid + & \text{Probing hash table with set elements} \\ (hc_t + hi_t) k_R \mid R \mid + & \text{Constructing hash table for } R_B \text{ during join} \\ (hc_t + hp_t) \sigma \mid R \mid & \text{Probing join hash table with qualified tuples} \end{cases} \tag{22}$$

For queries with the set in the result, the cost is almost the same, except that the cost of writing the result tuples is included as in nested internal. The cost of disjunctive queries is almost identical as in nested internal.

Case 2: Indexed Nested External Representation

The space requirements for indexed nested external INE_{SPACE} is given by

$$INE_{SPACE} = \begin{cases} \| R_B \| + \| R_S \| + & \text{Sizes of the relations} \\ \frac{2K \times LP_{SPACE}}{(K-1)} & \text{Total size of the unnested index} \end{cases} \tag{23}$$

Query evaluation cost is very similar to nested internal representation except the additional cost of the join. The I/O cost $INECQ_{IO}$ that includes the cost of reading the base relation R_B for the join is given by

$$INECQ_{IO} = \begin{cases} \| R_B \| \, IO_{seq} + & \text{I/O cost of reading the base relation} \\ N log_K \, (LP_{SPACE}) \, IO_{rand} + & \text{Probing predicate elements in the index} \\ \sigma \mid R \mid IO_{rand} + & \text{Cost of fetching the tuples} \\ \sigma \left\lceil \frac{(A_{abc}+S_s)|R|}{P_S} \right\rceil IO_{seq} + & \text{Cost of writing the result} \\ (2\sigma + 1) \mid R \mid B_t & \text{Fetching tuples from buffer pool} \end{cases} \tag{24}$$

The CPU cost $INECQ_{CPU}$ of evaluating the join and probing the index is given by

$$INECQ_{CPU} = \begin{cases} N log_2 \, (K) \, log_K (LP_{SPACE}) cp_t + & \text{Searching index pages} \\ & \text{using binary search} \\ N\sigma \mid R \mid cp_t + & \text{ANDing or merging TIDs} \\ (hc_t + hi_t) \mid R \mid + & \text{Constructing hash table} \\ & \text{for } R_B \text{ during join} \\ (hc_t + hp_t)\sigma \mid R \mid & \text{Probing join hash table} \\ & \text{with qualified tuples} \end{cases} \tag{25}$$

As expected, the cost evaluating queries using nested external and its index variant are very similar to nested internal and its index variant except for the additional cost of a join.

2.8 Summary

We have outlined three representations for storing sets: nested internal (the set elements are grouped and stored along with rest of the attributes), nested external (set elements are grouped and stored in a separate relation and connected back to the base tuple using keys and foreign keys) and unnested external (the set elements are unnested and stored in separate relation and using a foreign key that connects back to the base tuple). Further, we gave formulas for the analytical cost of conjunctive and disjunctive queries and their variants of projecting sets as a part of the result.

Chapter 3

Performance Evaluation of Set Storage Representations

3.1 Introduction

In this chapter, we evaluate the performance of various alternatives for representing sets in database systems: the nested internal, nested external, unnested external, indexed nested internal, indexed nested external and indexed unnested external representations. We conducted experiments both with our analytical model and with an implementation in an object relational system — Paradise [PYK$^+$97]. These experiments confirm the validity of the analytical model. Further, they conclude that the nested representations and its indexed variants provide substantial speedup over the unnested representations for conjunctive and disjunctive queries.

3.1.1 Chapter Organization

This chapter is organized as follows. Section 3.2 describes the data distributions used in our experiments. Section 3.3 presents the results of the analytical experiments: varying the average set cardinality, varying the selectivity, varying the number of elements in a predicate, and varying the size of a set element. Section 3.4 gives a short introduction to Paradise O/R DBMS. Section 3.5 describes the implementation of set representations in Paradise. The experimental schema and queries used in experiments are shown in Section 3.6. The results of various experiments are presented in Section 3.8. Finally, the results are summarized in Section 3.9.

3.2 Data Distributions and Experiments

Experiments are carefully chosen to identify the tradeoffs involved in each representation when different queries are executed. All experiments measure the response time for the execution of the query. The data distribution and the nature of the query executed influence the response time. We use the following parameters to characterize the set data distribution since their impact on response time of the query is more interesting.

- average set cardinality
- size of set element

Similarly, set queries are characterized by the following parameters:

- selectivity
- number of set elements in the predicate
- whether set is projected in the result or not

The set of experiments consisted of varying one of the parameters: the set cardinality, size of the set element, selectivity and number of elements in the predicate while keeping the others fixed.

Notation	Description	Value
$\mid R \mid$	Number of tuples in relation R	10,000
A_{abc}	Average size of the attributes a, b and c in relation R	68 bytes
S_s	System space overhead for storing a tuple	16 bytes
K_s	Space required for a key for external representations	4 bytes
B_t	CPU cost of accessing a tuple from the buffer pool	0.03574 ms
TID_s	Size of the tuple identifier	12 bytes
PID_s	Size of page identifier	4 bytes
IO_{seq}	Cost of a sequential I/O	0.665 ms
IO_{rand}	Cost of a random I/O	2.39 ms
P_s	Page size	4096 bytes
TA_t	Tuple assembly cost if the tuple is scattered across multiple pages	0.2812 ms
hi_t	Cost of inserting an element into hash table	0.0005 ms
hc_t	Cost of computing the hash function on an element	0.0010 ms
hp_t	Cost of probing an element in the hash table	0.0005 ms
cp_t	Cost of comparing two set elements	0.0005 ms
M	Amount of memory available in the system	32 MB

Table 2: Values Assigned to Analytical Parameters

3.3 Analytical Experiments

In this section, we present results from experiments with the analytical model for conjunctive queries with no sets in the results. Analytical experiments are plots of the equations for query evaluation cost derived in Chapter 2. Table 2 shows the values for various parameters used in the analytical model. These values are derived from the observed costs in Paradise. We varied four parameters in our experiments as described earlier.

3.3.1 Experiment 1: Varying Set Cardinality

In this experiment, we varied the cardinality of the set. Figure 5 shows the prediction of the analytical model using the parameters shown in Table 3.

As the cardinality is varied, the non-indexed representations showed an increase in

Set Cardinality	Selectivity	Number of Elements in the Predicate	Size of Set Element
10-100	10%	6	20

Table 3: Analytical Model Parameters for Varying Set Cardinality

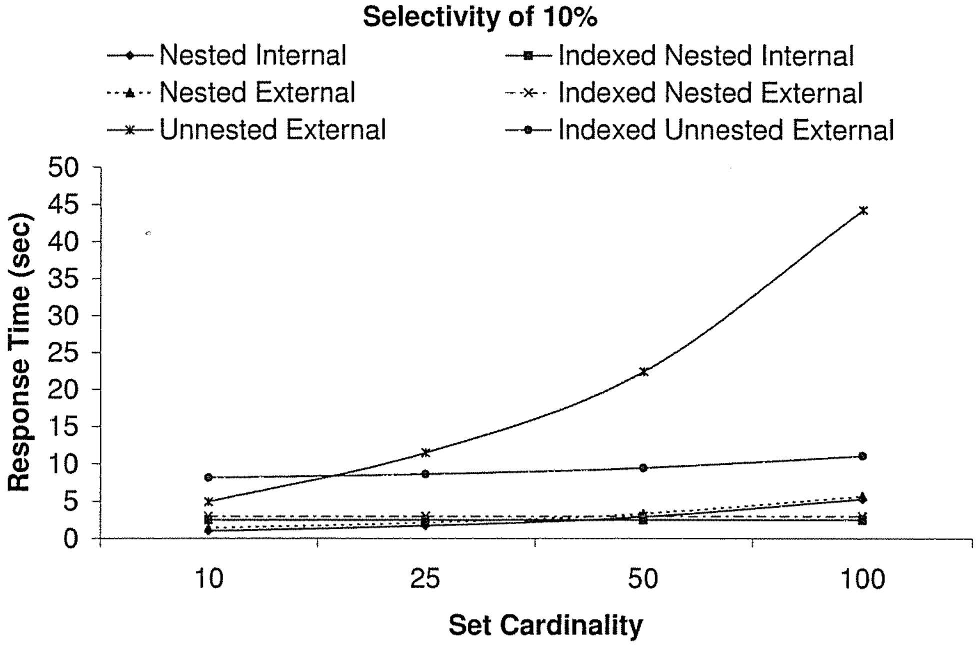


Figure 5: Analytical Model — Varying Set Cardinality for Selectivity of 10%

response time, since the number of pages the relation occupied increases with set size. Further since cardinality is increased more elements have to be examined to verify the predicate. Unnested external shows a sharp increase after a set cardinality of 25. This is because the number of tuples in the set table doubles and hence the cost of fetching, scanning and evaluating these tuples increases.

The nested indexed representations showed a small increase in response time. This is because of two reasons:

- The number of pages occupied by the leaf level of the index increases, thereby requiring longer seeks to get to the same set of elements.

- The number of pages occupied by the relation also increases thereby reducing the likely-hood of finding the result tuples in a page.

The performance of indexed unnested external representation is an order of magnitude slower than the nested ones. This is because the query involves many joins and simultaneous probing of the index by all joins. This leads to costly disk seeks.

At smaller cardinalities (< 40), the non-indexed nested representations performed better. This is because the cost of sequential scanning and evaluating the tuples in the entire relation is less than cost of random I/Os for probing in the index and fetching the result tuples from the relation. On the other hand, the unnested external representation performed better only at cardinalities less than 20.

3.3.2 Experiment 2: Varying Selectivity

In this experiment, we varied the selectivity of the predicate. Figure 6 shows the prediction of the analytical model obtained by varying selectivity using the parameters specified in Table 4.

Set Cardinality	Selectivity	Number of Elements in the Predicate	Size of Set Element
100	0.01%-50%	6	20

Table 4: Analytical Model Parameters for Varying Selectivity

As the selectivity is increased, the non-indexed representations shows a smaller increase in response time since the number of tuples written back to disk increases. Indexed representations do not do as well as non-indexed representations at higher selectivities since the number of tuples fetched increases and each fetch is a random I/O. However,

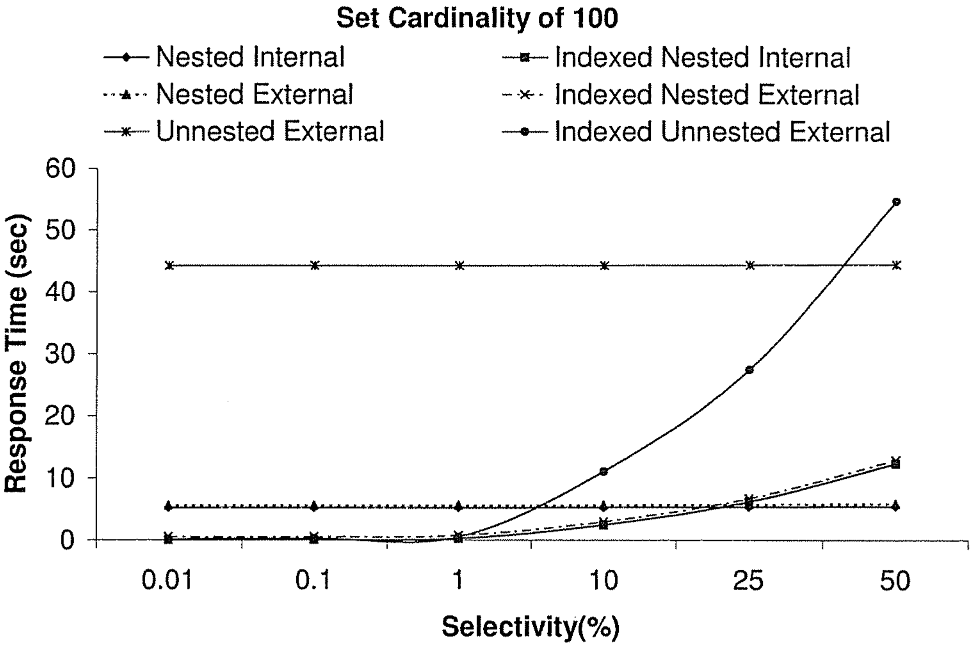


Figure 6: Analytical Model — Varying Selectivity for Set Cardinality of 100

the useful range of selectivities for use with an index is higher for the indexed nested representations (30%) than for the unnested representation (5%).

3.3.3 Experiment 3: Varying Number of Elements in Predicate

In this experiment, we varied the number of elements in the predicate. The other parameters are shown in Table 5. The prediction by the analytical model is shown in Figure 7.

Set Cardinality	Selectivity	Number of Elements in the Predicate	Size of Set Element
100	10%	1-6	20

Table 5: Analytical Model Parameters for Varying Number of Elements in the Predicate

When the number of elements in the predicate is varied, the nested representations and their indexed counterparts showed almost constant response times. This is because in

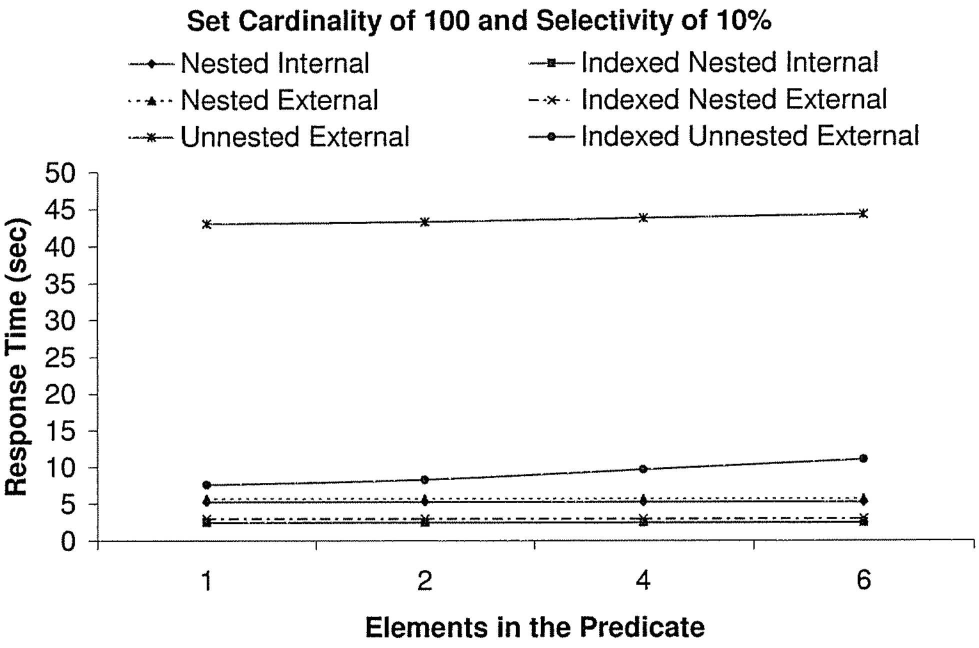


Figure 7: Analytical Model — Varying Number of Elements in Predicate

the nested representations the entire predicate is evaluated as a group using a hash table. When the set instance is scanned, each set element is hashed and probed (in most cases, the probe will fail and hence the comparison with a predicate element will not occur). The scan is also terminated prematurely if the predicate is satisfied early which leads to faster performance at higher selectivities. However, in the case of unnested external representation each predicate element has to be compared with every set element. This includes the cost of explicity comparing the element with all the predicate elements. If the predicate evaluation includes short circuiting, then each element will be compared with half of the predicate elements. Hence the cost of unnested external increases as the predicate elements are increased. On the other hand, the increase in the indexed unnested external representation performance is due to additional joins and the associated increase in the number of traversals of a B-Tree leading to disk seeks.

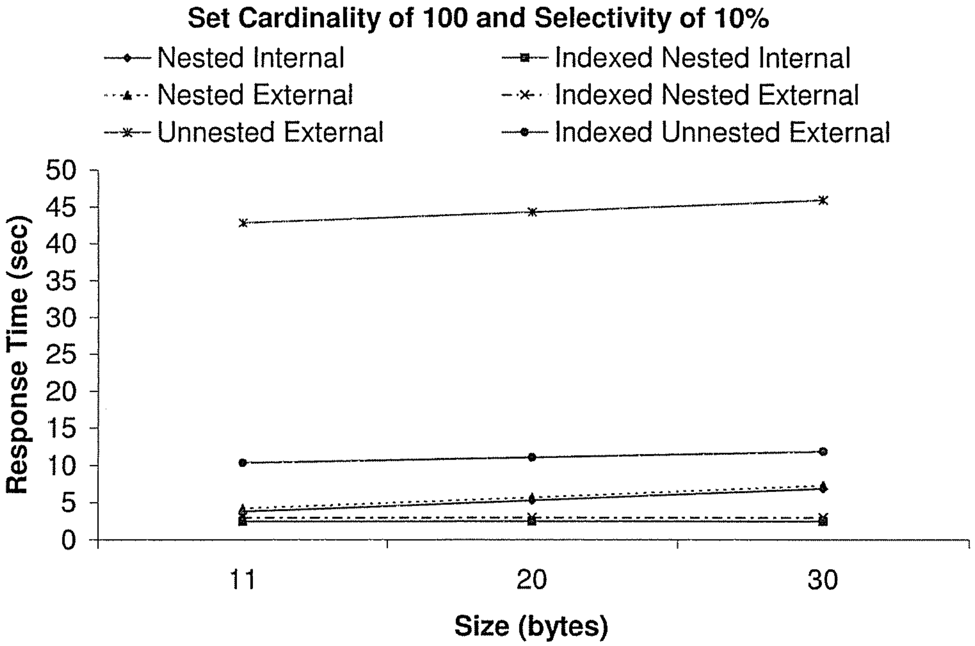


Figure 8: Analytical Model — Varying the Size of Set Element

3.3.4 Experiment 4: Varying Size of Set Element

In this experiment, we varied the size of the element. The parameters are summarized in Table 6. The prediction by the analytical model is shown in Figure 8.

Set Cardinality	Selectivity	Number of Elements in the Predicate	Size of Set Element
100	10%	6	11-30

Table 6: Analytical Model Parameters for Varying Size of Set Element

When the size of set elements is varied, the indexed versions showed almost constant (a very small rate of increase) response time since the range of sizes considered did not increase the height of B-Tree. On the other hand, the response time of the non-indexed representations increased linearly because of the increase in number of the I/Os as the number of pages in the relation increased.

Next, we will discuss the issues that arise when implementing the different representations in an O/R DBMS. Further, we will describe our results on each of these implementations.

3.4 Paradise O/R DBMS

Paradise is a shared nothing parallel object-relational database system developed at University of Wisconsin [PYK$^+$97]. It supports standard attribute types such as integers, floats, date and time. Paradise deviates from the regular relational systems by providing a set of built-in spatial types, including point, polygon, poly-line and circle as well as other additional types for storage and manipulation of images. It also provides a compile time type mechanism for adding new types and extending existing types. All these types can be used in any arbitrary combination when defining table types. These types are implemented as abstract data types (ADTs). Paradise ADTs are inherited from a base ADT, which provides common methods that every ADT must implement. These common methods provide the functionality for external input and output, conversion between memory and database representations, and the standard comparison and assignment operators. In addition to these common methods, each ADT has its own type specific methods.

3.5 Implementation

To support nested representations, we implemented a set ADT in Paradise. The set ADT implements a number of set-oriented methods, including: *create-iterator* and *set comparison operator.* Create iterator returns an iterator over the elements of the set. Set comparison operators are implemented by type specific methods that are invoked by the query engine when comparison and assignment are performed on sets. Specific details on each implementation are described below.

- **Nested Internal Representation:** It uses the the set ADT directly. Queries are issued by invoking the appropriate methods in the set ADT.

- **Nested External Representation:** We manually created two tables: the base table and the set table. The set-valued attribute was stored in the set table in the same way that the set-valued attribute was stored in the nested internal implementation. This shows that nested external representation does not require any additional effort if the nested internal representation is available. Again, the queries were issued manually using a combination of method invocation on the set ADT and relating the base tuples using a join.

- **Unnested External Representation:** Recall that the unnested external representation does not require engine-level changes. For this reason, we were able to "implement" the unnested external representation by explicitly creating two tables as in the nested external representation, and then hand "compiling" set-valued queries into their relational equivalents.

- **Indexed Nested Internal Representation:** For implementing indexed nested representations, we implemented an unnested index on the set ADT that first unnests the set instance using the iterator, next stores set element and TID pairs in a temporary table, then sorts the temporary table and loads it into B-Tree using the bulk loading mechanism. We directly created the unnested index on the set-valued attribute in the base table. In this case, the queries are fed into the execution engine by explicitly specifying the plan.

- **Indexed Nested External Representation:** The same nested index is used as in indexed nested internal representation but the index was created on the set table. Again, the queries are fed as plans.

- **Indexed Unnested External Representation:** We created two indices for the unnested external representation: a clustered index on the foreign key attribute, and an unclustered index on the set element in the set table. These two indices are functionally equivalent to a join index on the two tables. The plans for queries used indexed nested loop join.

3.6 Experimental Schema and Queries

We used the following schema for our experiments. The schema describes *people* and the set of *movies* they have watched.

Moviegoer(name, street, city, state, zipcode, movies)

The queries we ran represented three main classes of set-valued queries: conjunctive queries, disjunctive queries and queries that do not refer the set-valued attribute.

- **Conjunctive Queries:** In this class, we ran two queries: one that does not contain set in the result and the other contains set in the result.

```
SELECT m.name, m.street, m.city, m.state, m.zipcode
FROM Moviegoer m
WHERE ('movieA50061', 'movieA50062') SUBSET OF m.movies          (Q.24)
```

```
SELECT m.name, m.street, m.city, m.state, m.zipcode, m.movies
FROM Moviegoer m
WHERE ('movieA50061', 'movieA50062') SUBSET OF m.movies          (Q.25)
```

- **Disjunctive Queries:** In this class also, we ran two queries: one that does not contain set in the result and the other contains set in the result.

```
SELECT m.name, m.street, m.city, m.state, m.zipcode
FROM Moviegoer m
WHERE 'movieA50061' IN m.movies OR 'movieA50062' IN m.movies          (Q.26)
```

```
SELECT m.name, m.street, m.city, m.state, m.zipcode, m.movies
FROM Moviegoer m
WHERE 'movieA50061' IN m.movies OR 'movieA50062' IN m.movies          (Q.27)
```

- **Queries Not Referencing Set Valued Attribute:** In this class, we ran a select and a self join query. These queries do not reference the set-valued attributes and they are included to investigate the impact of the representations on normal relational query processing.

```
SELECT m.name, m.street, m.city, m.state, m.zipcode
FROM Moviegoer m                                                      (Q.28)
```

```
SELECT m1.name, m1.street, m1.city, m1.state, m1.zipcode
FROM Moviegoer m1, Moviegoer m2
WHERE m1.id = m2.id                                                   (Q.29)
```

For unnested external, we use the *disjunctive groupby* approach as specified in Chapter 2. For indexed unnested external, we use the *cascaded joins*. The reasoning is that with cascaded joins unnested external requires multiple joins with set table and hence might be intolerably slow. The choice of running cascaded joins with indexed nested external is to understand whether such a plan is useful on a restricted set of selectivities of the query.

3.7 Experimental Setup and Data Generation

The data for our experiments was synthetically generated in such a way that each set had the same number of elements (this number is varied in some of the experiments below). In addition, the data generator also ensured that it was easy to generate predicates over the set-valued attributes with any desired selectivity. The data was uniformly distributed. While it would certainly be interesting to experiment with varying numbers of elements per set and various distributions over the elements of the set, the main conclusions from our experiments are so clear that such experiments would be unlikely to significantly affect them.

The average size of each tuple in the relation *Moviegoer* was 68 bytes without including the instance of the set-valued attribute and the average size of each element in the set movies was 20 bytes. The total number of tuples in the *Moviegoer* relation was 10,000. Our experiments were run on Intel 333 Mhz Pentium processors with 128MB of main memory running Solaris 2.6. We used raw disks providing a measured I/O bandwidth of 5 MB/sec and with an available capacity of 4GB. We used a database buffer pool size of 32MB. Each query was run against a cold database by flushing the buffer pool before each experiment.

In the following sections, we present the results of the experiments and interpret them. We explored a large portion of the performance space. However, here we present only the representative results. We chose to present the results for only conjunctive queries with no set in the result since the other queries follow a similar trend.

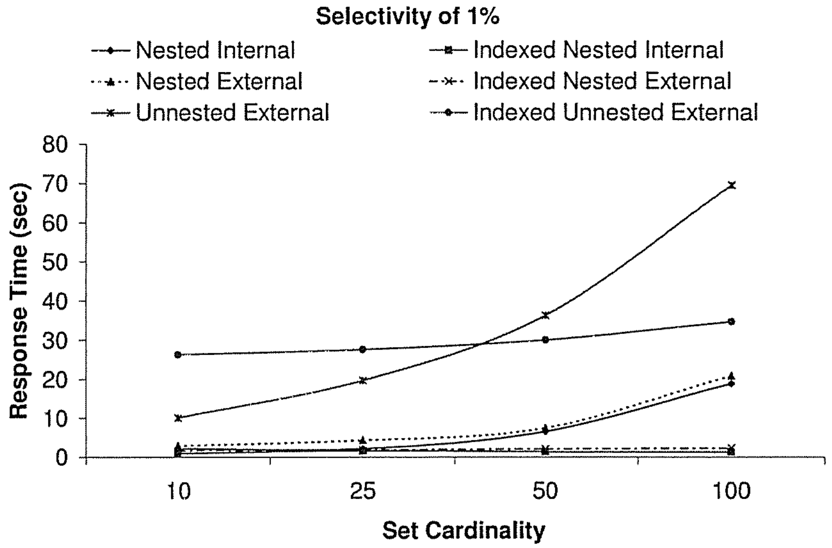


Figure 9: Varying Set Cardinality with Selectivity of 1%

3.8 Implementation Experiments

3.8.1 Experiment 1: Varying Set Cardinality

In these experiments, we study the effect of varying set cardinality. The parameters for these experiments are shown in Table 7. The results are shown in Figure 9.

Set Cardinality	Selectivity	Number of Elements in the Predicate	Size of Set Element
10-100	1%	6	20

Table 7: Experiment Parameters for Varying Set Cardinality

When increasing the number of set elements, all the representations show an increase in cost. This is expected since an increase in the number of set elements requires examining more tuples and/or more elements. Among the non-indexed representations, the nested internal representation provides the best performance since it involves only a scan and the cost of predicate evaluation as predicted by the analytical model. The performance of nested external representation closely followed the nested internal representation within a range of 10%. This constant factor is due to the join involved to connect the qualified nested set table tuples with the base table tuples. On the other hand, the rate of increase in response time is larger for the unnested external representation (especially after 50) when compared with the other representations. A detailed investigation of this behavior is presented later in the section.

In general, the indexed representations performed 50% faster than their non-indexed counterparts, except for the indexed unnested external representation. Indexed unnested external increases when the set cardinality is more than 50. This is because the query contains 6 joins and the index is probed by each join which involves random seeks when they are executed in a pipeline fashion.

The non-indexed representations perform better than the indexed representation when the cardinality is low. This is because the number of pages the relation spans is much smaller and the cost of reading the relation sequentially is less expensive than the disk seeks involved in fetching the tuples from the index.

Performance of Unnested External Representation

To investigate the deteriorating performance of the unnested external representation, we measured the individual costs that contributed to the overall response time. The results are plotted in Figure 10. The cost break down for nested internal representation is plotted in Figure 11 for comparison.

As can be seen from the figures, the unnested external representation suffers from the "cardinality explosion" problem. When the cardinality of the set is increased, the number of tuples in the set relation is increased by the product of the number of base

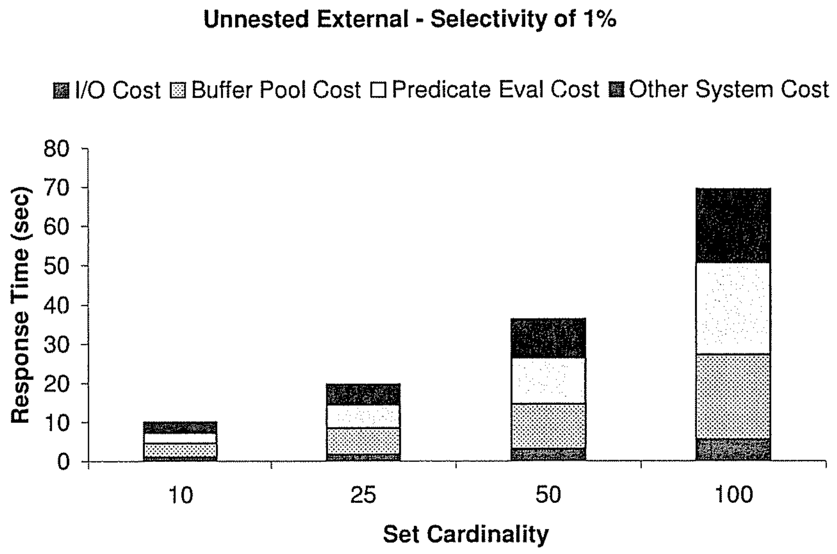


Figure 10: Cost Breakdown for Unnested External

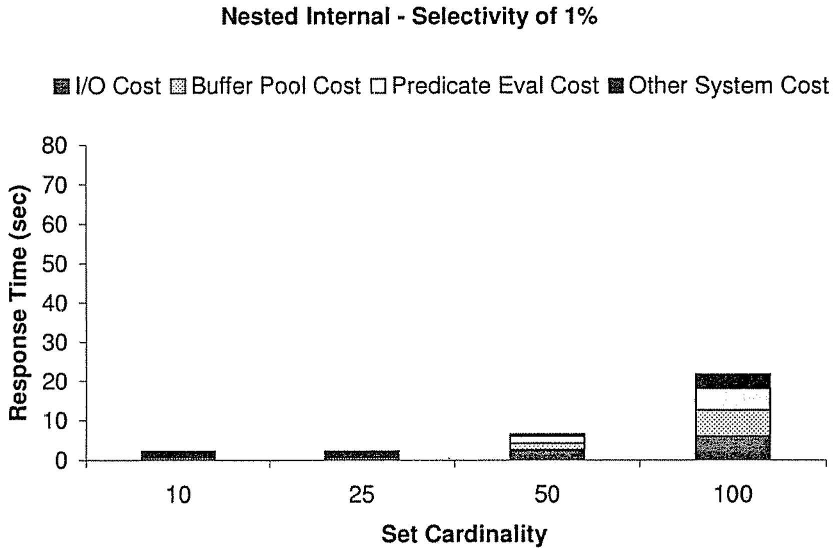


Figure 11: Cost Breakdown for Nested Internal

tuples and the cardinality of the set. Further the number of tuples entering the join is increased since the evaluated predicate is a disjunction. This disjunction passes up tuples belonging to sets which might not contain all the predicate elements to qualify in the final result. Such an increase in tuples adds the following costs:

- The cost of fetching more tuples from buffer pool while scanning the set table. This cost includes the cost of pinning the page, locating the record in the slotted page and extracting it.
- The cost of predicate evaluation during scanning the set table, joining of the base table tuples with set table tuples and the final "groupby".

On the other hand, the nested internal representation had fewer tuples that needed to be fetched from buffer pool. Further, the cost of predicate evaluation is not as high since the predicate is evaluated on the whole set. In order for the unnested external representation to be more competitive, the cost of fetching tuples from buffer pool and the cost of predicate evaluation cost have to be improved.

3.8.2 Experiment 2: Varying Selectivity

These experiments varied the selectivity of the predicate on the set. The parameters used in these experiments are summarized in Table 8 and the results are plotted in Figure 12. A closer version of this figure is shown in Figure 13 which excludes two data points of indexed unnested external.

The non-indexed nested representations showed a very small increase in time. This is to be expected since more tuples must be written out as part of the result. However, the unnested external representation showed a larger increase in response time above a selectivity of 1%. This increase is due to the increase in the number of tuples feeding

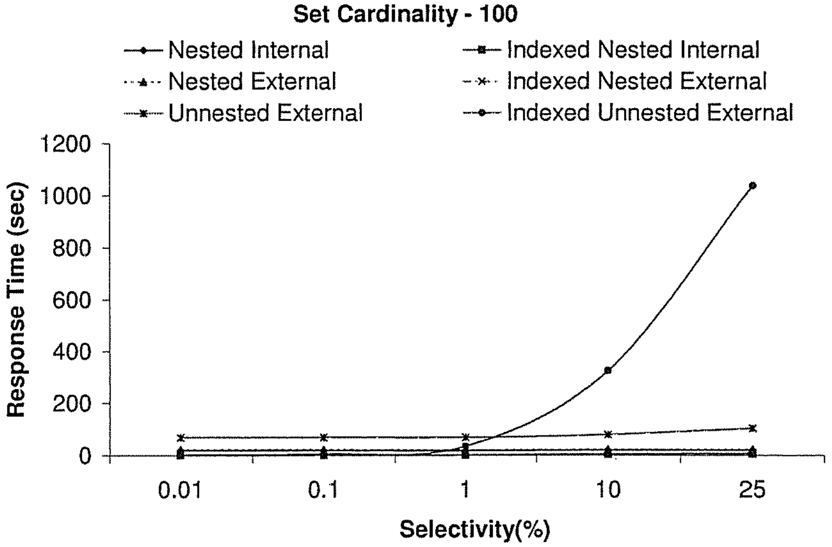


Figure 12: Varying Selectivity for Set Cardinality of 100

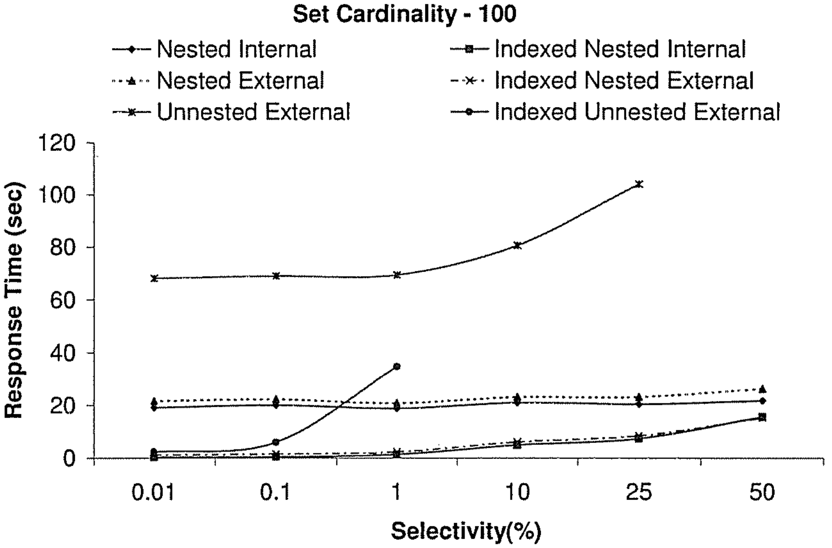


Figure 13: Detailed Version of Varying Selectivity for Set Cardinality of 100

Set Cardinality	Selectivity	Number of Elements in the Predicate	Size of Set Element
100	0.01%-50%	6	20

Table 8: Experiment Parameters for Varying Selectivity

the join. This increase also affects the cost of "group by" since more tuples have to be examined. Further, the number of result tuples written to disk is also higher and contributed to the increase in response time.

The indexed representations provided fast performance at lower selectivities. However, the indexed representations catch up on their non-indexed counterparts at higher selectivities. As the selectivity increases, more probe of the index is required and more tuples are fetched from disk, causing increased random I/Os. The indexed unnested external starts increasing sharply at about 3%. This is due to the following factors:

- The query executes 6 joins and each join is probing the index simultaneously with other joins. Further each join filters only a subset of the predicate. Hence the number of tuples entering the lower most join is more. Because of this increased tuples, subsequent joins also see more tuples.

- The base table is joined early with set-table so the attributes in the result are passed to subsequent joins. This increases the cost substantially since at every join the tuples have to be split for predicate evaluation. At the end of join, they have to be assembled again and passed to the next join.

Because of aforementioned overheads, the usability range of indices in the unnested external representation is limited. The usability range of index on nested representations extends to 50% because the individual tuples are large and occupy a page on their own, causing scan time to go up as compared to their non-indexed counterparts.

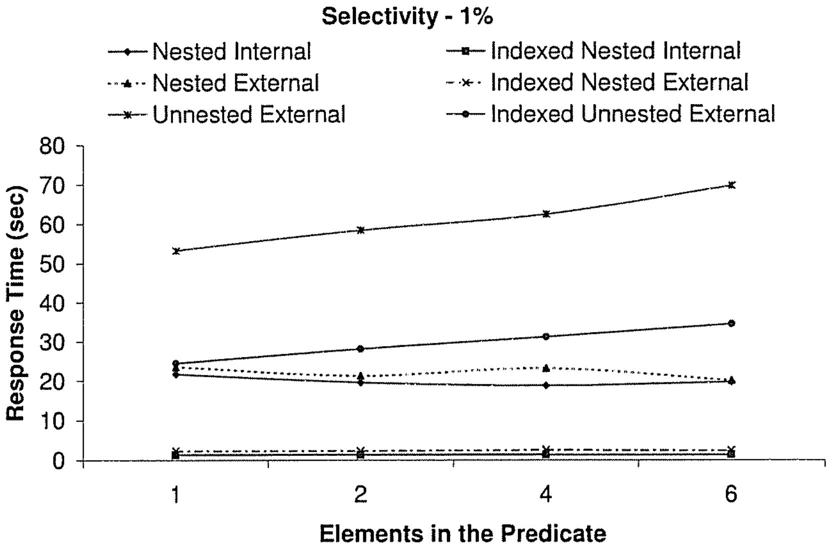


Figure 14: Varying Number of Elements in the Predicate with Selectivity of 1%

3.8.3 Experiment 3: Varying Number of Elements in Predicate

These experiments studied the effect of increasing the number of elements in the selection predicate. The parameters chosen for these experiments are shown in Table 9. The results are plotted in Figure 14.

Set Cardinality	Selectivity	Number of Elements in the Predicate in the Predicate	Size of Set Element
100	1%	1-6	20

Table 9: Experiment Parameters for Varying Number of Elements in the Predicate

Varying the number of elements in the predicate does not affect nested representations since the entire predicate is evaluated as a group, by examining all the set elements. When the predicate fails, the entire set would have been scanned. If the predicate is satisfied early, then the scan of the set elements is terminated immediately. Since the selectivity is 1%, the number of tuples that satisfy the predicate will be less and hence the impact of early termination is less. Hence, there is not much change in response time. Further, since a hash table is used for predicate evaluation, explicit comparison of elements is not required unless the hash values are equal. It provides substantial savings especially if the compared elements are large types — strings, images, documents etc.

On the other hand, the unnested external requires comparing every element in the predicate with all the tuples in the set table since the query rewrite contains a disjunction. If the evaluation of disjunction is short-circuited, then on the average every tuple in the set table is compared with half the number of elements in the predicate. Hence as the number of elements in the predicate increases, the response time increases linearly.

Since there is no short circuited evaluation across elements as in nested representation, one might argue that the set-table in unnested external could be sorted. Sorting ensures that elements from a single set are clustered. The predicate evaluation takes advantage when the set predicate is satisfied early, it can skip rest of the elements and proceed to the next set. However, this will require changes in the engine to implement a skipped sequential scan and a specialized predicate evaluator.

The indexed nested representations showed a very small increase in response time. This increase is due to the increased number of probes and the "anding" of the rids of the tuples. For indexed unnested external, increasing the number of elements in the predicate adds more joins to the query and the operators heavily use both the indices thereby increasing the disk seeks.

3.8.4 Experiment 4: Varying Size of the Set Element

In this experiment, we study the impact of varying the size of the set elements. The experiment parameters are shown in Table 10. The results are plotted in Figure 15.

Set Cardinality	Selectivity	Number of Elements in the Predicate	Size of Set Element
100	1%	1-6	11-30

Table 10: Experiment Parameters for Varying Size of the Set Element

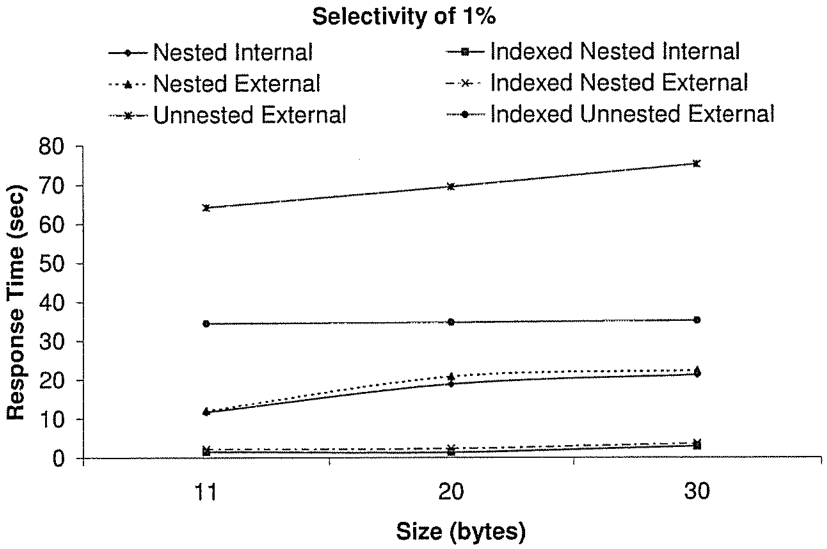


Figure 15: Varying Size of the Set Element for a Selectivity of 1%

All of the non-indexed representations show a gradual increase in response time since the number of pages the relation occupies increases as the size of the set element increases. However, there is an increase for nested representations from 11 bytes to 20 bytes since the number of pages increase by a factor of two. But from 20 to 30 bytes the performance of the nested representations is almost constant, because the number of pages remained the same. This is because a single tuple with elements of either 20 or 30 bytes can fit in a single page with the rest of the space remaining unused. All of the indexed representations showed constant performance since the number of probes and number of tuples were the same in all cases.

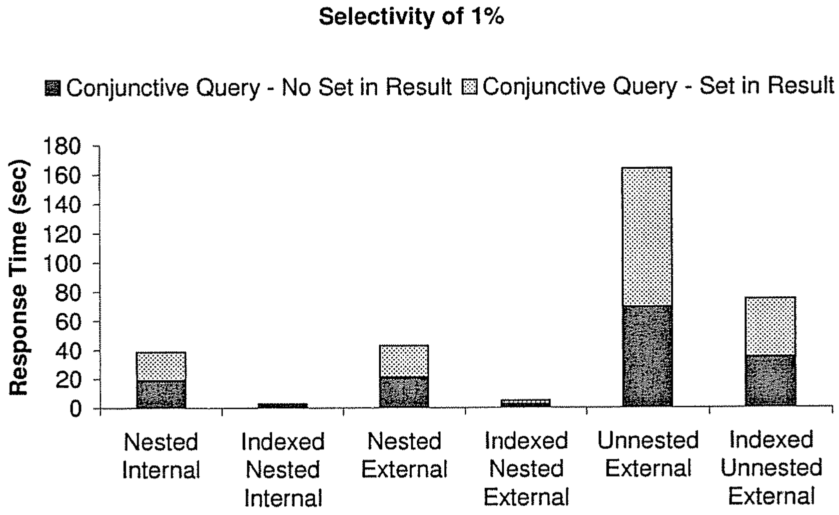


Figure 16: Comparing Conjunctive Query Variants

3.8.5 Experiment 5: Comparing Conjunctive Query Variants

In this section, we compare conjunctive queries both with the "set in the result" and "not in result". The parameters used in the experiment are shown in Table 11. The results are shown in Figure 16.

Set Cardinality	Selectivity	Number of Elements in the Predicate	Size of Set Element
100	1%	6	20

Table 11: Experiment Parameters for Comparing Conjunctive Query Variants

This graph shows that the percentage increase for grouping the base tuples and its set elements (projection) is slightly high for the unnested external representation and its indexed variant, while for the nested representations and their indexed variants it is lower. The increase in the nested internal representation is due to the CPU cost associated with fetching tuples from a sequential scan and writing the projected set-valued attribute.

However, the write cost is small, since for indexed nested internal representation, this comes only from writing the tuple. The increase in unnested external representation is due to an additional join that involved a scan of the large set relation and the cost of writing more tuples.

On the other hand, for the nested external representation, the increase is much lower in spite of almost the same amount of data being passed through the join as in the case for the unnested external representation. This is because in this representation, many fewer tuples carry the same amount of data. However, the join cost is very low.

3.8.6 Experiment 6: Overhead

These experiments were run in order to measure the overhead associated with the nested internal representation. This is of interest especially if the predominant query workload does not use the set-valued attribute. For these experiments, we used the parameters shown in Table 12. We ran the simple select and self join queries as described in Section 3.6, and the results are plotted in Figure 17 and Figure 18.

Set Cardinality	Selectivity	Number of Elements in the Predicate	Size of Set Element
100	100%	NA	20

Table 12: Experiment Parameters for Queries that do not refer the Set Valued Attribute

For the select query, the time for the nested internal representation is dominated by the extra I/O cost, which is proportional to the total size of all instances of the set. For the join query, the extra cost is linear in the combined cost of the set instances from both relations. For the unnested external and nested external representations, the cost is the same independent of the size of the set since only the base relation was examined.

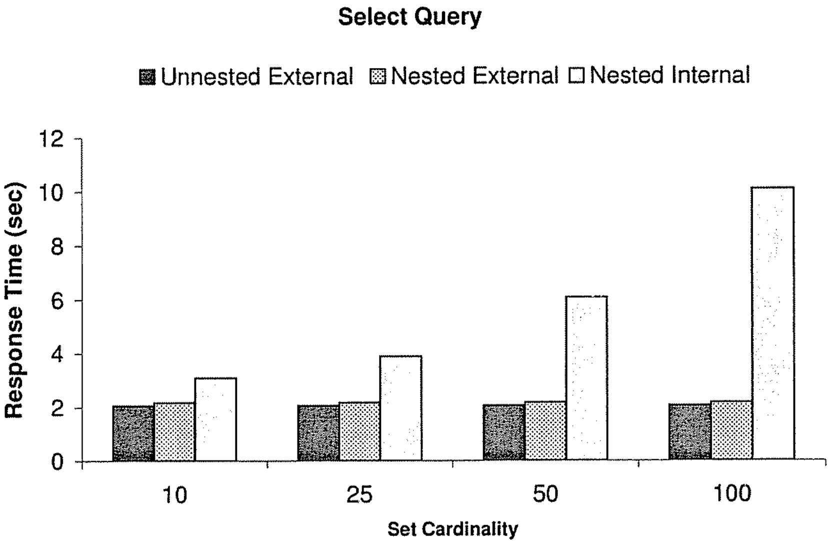


Figure 17: Overhead for Selection Queries that do not refer to the Set Valued Attribute

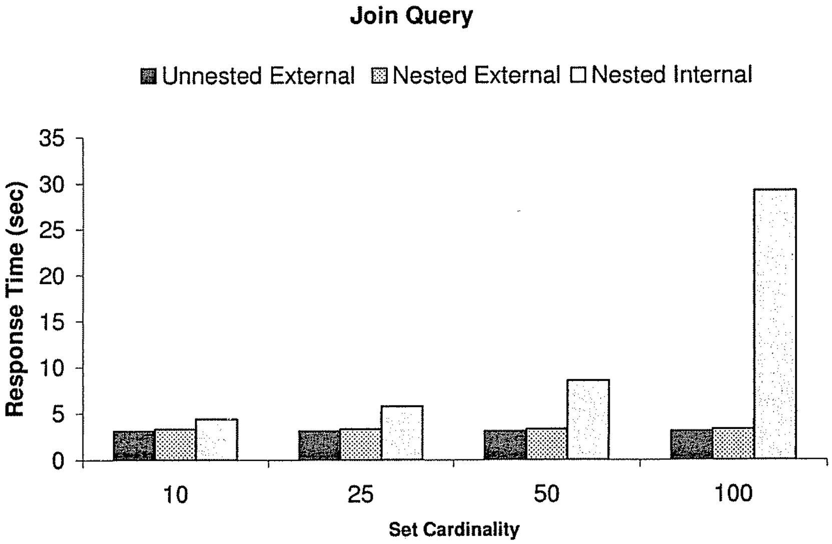


Figure 18: Overhead for Join Queries that do not refer to the Set Valued Attribute

3.9 Summary

The efficient implementation of set-valued attributes is an important challenge facing the developers of "Universal Servers" or other systems that support SQL4. We analyzed the space of representations available for storing set-valued attributes and developed a detailed analytical model. We verified the analytical model with an implementation in Paradise. In all, the experiments conclude that the nested representations perform better than the unnested representations.

The performance of the unnested representations suffers from the high cost of evaluating queries even for simple conjunctive and disjunctive queries. This is because the cost of fetching the tuple from buffer pool and processing the predicate is much expensive since each set element is stored as a separate tuple. These costs are proportional to the number of set elements, unlike nested representations, in which it is proportional to the number of sets.

The indexed variants of these representations facilitated faster probing for disjunctive and conjunctive predicates, however, the usability of indices is limited to particular range of selectivities. Even though the nested representations perform better for set queries, they have the drawback of requiring additions and modifications to the storage and query evaluation engine of a relational database system. However, the modifications translate into large improvements in performance.

Chapter 4

Set Containment Join Algorithms

4.1 Introduction

We now turn away from set representation alternatives to consider the implementation of a particularly challenging operation over set-valued attributes, the set containment join.

Many real world queries can be easily expressed using set containment joins. Consider the following example that involves two relations.

4.1. Example.

Student (sid, {courses})

Courses (cid, {pre-requests})

where the attributes {courses} and {pre-requests} are set-valued attributes. A frequently asked query on this data set might be to find the courses that the students are eligible to take. Such a query can be easily expressed as a set containment join as follows:

```
SELECT s.sid, c.cid
FROM Student s, Courses c
WHERE c.pre-requests ⊆ s.courses                                        (Q.30)
```

Another motivating example arises from the web. Consider a simple relation that describes documents and the set of hyper-links that point to them:

4.2. Example.

Document(did, {hyper-links-in}, actual-document)

Suppose document d_1 is defined to be more important than d_2 if d_1 is linked-to by a superset of the documents that link to d_2. We can find pairs of documents (d_1, d_2) for which d_1 is more important than d_2 with the following query:

SELECT d_1.did, d_2.did

FROM Document d_1, Document d_2

WHERE d_2.hyper-links-in $\subset$ d_1.hyper-links-in $\quad (Q.31)$

The algorithms available for implementing set containment joins depend upon how set-valued attributes are stored in the database. As described in Chapter 2, sets can be stored in the *nested internal representation* (set elements are stored together along with the rest of the attributes) or the *unnested external representation* (set elements are scattered and stored in a separate relation).

To the best of our knowledge, current commercial O/R DBMS use the unnested external representation. Since the unnested external representation reduces to standard SQL2 relations under the covers, set containment joins on the unnested external representation can be evaluated by rewriting the queries into SQL2 (with no sets) and evaluating these rewritten queries. On the other hand, with the nested internal representation, the most obvious algorithm for evaluating set containment joins is nested loops. Two questions immediately arise: (1) Are there better algorithms than nested loops? (2) How do these algorithms compare in efficiency with the "rewrite in SQL2" approach that is most logical for the unnested external representation?

This chapter attempts to answer these questions by proposing a new partition-based join algorithm for set containment joins, which we call PSJ. Partition-based algorithms certainly dominate join algorithms in scalar and spatial domains, so it is natural to suspect that a partition-based algorithm will be the algorithm of choice for set containment joins.

In this chapter, we present the new algorithm PSJ for set containment joins. The next chapter includes an extensive performance study of three set containment algorithms: the traditional SQL approach on the unnested external representation, and signature nested loops and PSJ on the nested internal representation. Our experience with an implementation in the Paradise object-relational database system [PYK$^+$97] shows that PSJ yields significant speedup over both the SQL-based approach and signature nested loops. An added benefit of this algorithm is that, like all partition-based algorithms, it is trivially parallelizable. Finally, our results present more ammunition in the case for storing sets in the nested internal form, since PSJ and even signature nested loops outperform the rewritten queries over the unnested external representation.

4.1.1 Chapter Organization

The rest of the chapter is organized as follows. Section 4.2 defines the problem of set containment and the notation used in the chapter. The SQL approach and signature nested loops joins are explained in detail in Section 4.3. The need for partition based algorithms is justified in Section 4.4. The partition based set join algorithm is outlined in Section 4.4.2. Our conclusions are presented in Section 4.5.

4.2 Problem Definition and Notations

For the rest of the chapter, we consider the two relations $R(a, \{b\})$ and $S(c, \{d\})$ containing the set-valued attributes $\{b\}$ and $\{d\}$ respectively. Since set is a type constructor, attributes b and d can be of an arbitrary type, and we assume that these types provide an equality predicate that determines the equivalence of two set elements. Also, we do not assume any order among the set elements. The set containment join $R \bowtie_{\{b\} \subseteq \{d\}} S$ pairs tuples in relation R and S such that $\{b\}$ is subset of $\{d\}$. Table 13 describes the notation used in the rest of the chapter.

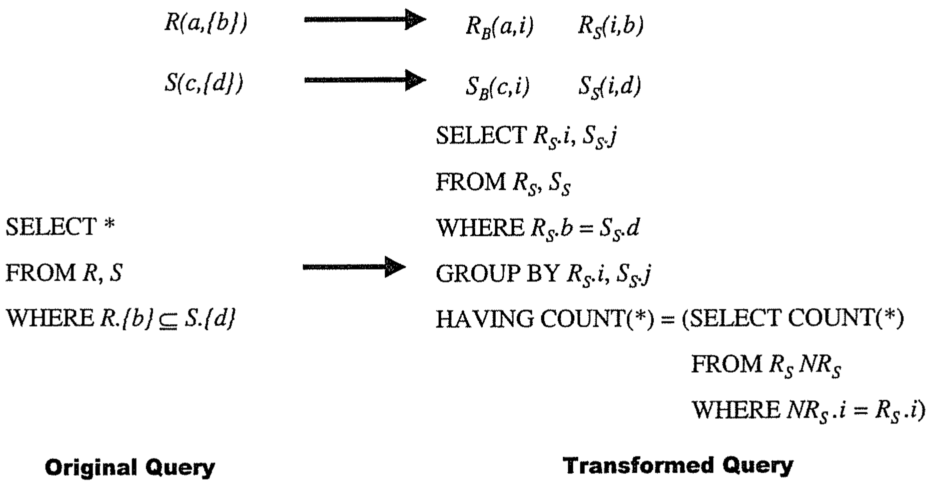
Notation	Description
$\mid\mid R \mid\mid$	Number of pages of R
$\mid\mid S \mid\mid$	Number of pages of S
$\mid R \mid$	Relation cardinality of R (# of tuples)
$\mid S \mid$	Relation cardinality of S (# of tuples)
k_R	Average set cardinality of R
k_S	Average set cardinality of S
r_R	Replication factor of R
r_S	Replication factor of S
σ	Selectivity of $R \bowtie_{\{b\} \subseteq \{d\}} S$
f	False drops as a percent of $\sigma \mid R \mid\mid S \mid$
TID_s	Size of an rid (bytes)
P_S	Size of the data page (bytes)
h_c	CPU cost of hash computation
s_c	CPU cost of comparing signatures
IO_{seq}	Cost of a sequential I/O
IO_{rand}	Cost of a random I/O

Table 13: Description of Notation Used

4.3 Previously Proposed Algorithms

4.3.1 Join Algorithms for Unnested External Representation

If sets are stored in the unnested external representation, set containment joins can be expressed and evaluated using standard SQL2 constructs. This approach is important to study, because (a) it is the simplest to add to any RDBMS, and (b) perhaps because of (a), to our knowledge the commercial O/R DBMSs all use this approach. As we

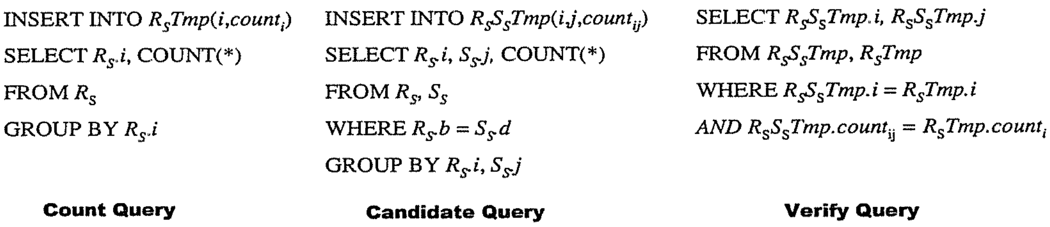


```
R(a,{b})   ──►  R_B(a,i)   R_S(i,b)
S(c,{d})   ──►  S_B(c,i)   S_S(i,d)

SELECT *                 SELECT R_S.i, S_S.j
FROM R, S                FROM R_S, S_S
WHERE R.{b} ⊆ S.{d}  ──► WHERE R_S.b = S_S.d
                         GROUP BY R_S.i, S_S.j
                         HAVING COUNT(*) = (SELECT COUNT(*)
                                            FROM R_S NR_S
                                            WHERE NR_S.i = R_S.i)

Original Query           Transformed Query
```

Figure 19: Original and Transformed SQL Queries (excluding final joins for a and c)



```
INSERT INTO R_STmp(i,count_i)
SELECT R_S.i, COUNT(*)
FROM R_S
GROUP BY R_S.i

Count Query
```

```
INSERT INTO R_SS_STmp(i,j,count_ij)
SELECT R_S.i, S_S.j, COUNT(*)
FROM R_S, S_S
WHERE R_S.b = S_S.d
GROUP BY R_S.i, S_S.j

Candidate Query
```

```
SELECT R_SS_STmp.i, R_SS_STmp.j
FROM R_SS_STmp, R_STmp
WHERE R_SS_STmp.i = R_STmp.i
AND R_SS_STmp.count_ij = R_STmp.count_i

Verify Query
```

Figure 20: Magic Sets Rewriting

discussed in Chapter 2, in this representation, a relation with a set-valued attribute is decomposed into two relations. A set containment operation can then be expressed using SQL over these decomposed relations. If R and S are the two relations being joined, and R_S and S_S are the corresponding decomposed auxiliary set relations, then the original and transformed queries are shown in Figure 19.

The rewritten query joins the set relations and groups the pair of sets that have at least one element in common. Then, for each group, it checks whether the size of the group (which is the number of elements in common between the R set and the S set) is the same as the cardinality of the set in R. This query, as written, returns in the answer tuples only the pair of set ids that satisfy the containment. If any other attributes were included in the answer, additional joins would be required to extract them from the input relation.

The main problem with this approach is the efficiency of evaluation of this query. Since this is a correlated nested query, the nested query must be evaluated for each group. Since the number of groups is proportional to $\mid R \mid \times \mid S \mid$, the cost of evaluating the nested query naively can be prohibitively large. A possible optimization is to use magic-sets rewriting [SPL96] and transform the original query into the set of queries shown in Figure 20, thus evaluating the inner query only once (as opposed to once for every tuple produced by the outer block). Even after this transformation, each set element in R_S is compared with every other set element in S_S. Hence the evaluation of the block (join followed by group-by) is likely to be the dominating cost.

Our experiments show empirically that even this approach performs very poorly unless the set sizes and relation sizes are small.

4.3.2 Signature Nested Loops Algorithm for Nested Internal Representation

Nested loop algorithms for set containment fall into two broad categories: *naive nested loops* and *signature nested loops.* In naive nested loops, the set containment predicate is evaluated on pairs of sets for the entire cross product of R and S. As shown in [HM97], naive nested loops performs poorly, since it is very expensive to compute the set containment predicate for every pair of tuples. Hence we do not consider naive nested loops in the remainder of this chapter.

The signature nested loops algorithm proposed by [HM97] attempts to reduce the cost of evaluating the containment predicate by approximating sets using signatures and evaluating the join predicate by comparing these signatures. A signature is a fixed length bit vector that is computed by applying a function M iteratively to every element e in the set, and setting the bit determined by $M(e)$. If the containment predicate $s \subseteq t$ is to be satisfied for two signatures s and t, then the following condition is necessary: *For all bit positions that are set to 1 in signature s, the corresponding bits in signature t should be set to 1.*

However, this condition, while necessary for determining that two tuples join, is not sufficient, since signatures are only an approximate representation for the set (unless the signature length is equal to the size of the domain of the set). Hence, using signatures to evaluate a predicate will yield false drops. That is, two sets may have signatures that indicate containment, while the actual sets do not really satisfy the containment predicate. The actual sets must be examined to eliminate these false drops.

We are now ready to describe the signature nested loops algorithm. This algorithm operates in three phases: the *signature construction phase*, the *probing phase*, and the *verification phase*. During the signature construction phase, the entire relation R is scanned, and for every tuple $t_i \in R$, a signature s_i is constructed. A triplet (c_i, s_i, OID_i) is computed and stored in an intermediate relation R_{sig}; here c_i is the set cardinality and OID_i is the physical record identifier (rid) of the tuple. The same process is repeated for the relation S and an intermediate relation S_{sig} is created.

Next, the algorithm proceeds to the probing phase, where the tuples of R_{sig} and S_{sig} are joined. For every pair $(c_i, s_i, OID_i) \in R_{sig}$ and $(c_j, s_j, OID_j) \in S_{sig}$, two conditions must be verified (i) $c_i \leq c_j$ and (ii) $s_i \wedge s_j = s_i$, where $\wedge$ represents the bit-wise "and" of the two signatures. If both the conditions are satisfied, then the pair (OID_i, OID_j) is a possible candidate for the result. During the final verification phase, the tuples referred to in the candidate (OID_i, OID_j) pairs are fetched and the subset predicate is evaluated on the actual set instances, producing the final result.

The main issue in the signature nested loop join algorithm is reducing the number of false drops to minimize the cost of the verification phase. The false drop probability depends on the number of bits used in constructing the signature. The greater the signature length, the smaller will be the false drop probability. However, larger signatures lead to more bit comparisons per signature, thereby increasing the execution time of the probing phase. Hence, it is necessary that the chosen signature size be such that further increases in the number of bits do not significantly reduce the false drop probability.

4.3.3 Estimation of Signature Size

Estimating the signature size requires the computation of the false drop probability. The false drop probability P_{FD} is defined in [IKO93] and is given by

$$P_{FD} = \frac{falsedrops}{N - actualdrops} \tag{26}$$

where N is the total number of comparisons and *actualdrops* is the total number of qualified pairs of tuples (including the false drops.) Equation (26) can be rewritten as

$$P_{FD} = \frac{falsedrops}{N - resultsize - falsedrops} \tag{27}$$

Now *resultsize* can be expressed as $\sigma \mid R \mid\mid S \mid$, where σ is the join selectivity. The *falsedrops* can be expressed as a percentage of *resultsize* as $f\sigma \mid R \mid\mid S \mid$, where f is the tolerable false drop percentage. Now P_{FD} can be expressed as

$$P_{FD} = \frac{f\sigma \mid R \mid\mid S \mid}{\mid R \mid\mid S \mid - \sigma \mid R \mid\mid S \mid - f\sigma \mid R \mid\mid S \mid} \tag{28}$$

The false drop probability P_{FD} of a subset predicate between two sets of size k_R and k_S is derived in [IKO93] and is given by

$$P_{FD} = (1 - e^{-k_S/F_{SNL}})^{k_R} \tag{29}$$

Using equations (28) and (29), we can determine the the optimal signature length (F_{SNL}) as

$$F_{SNL} = \frac{-k_S}{ln\left(1 - \left(\frac{f\sigma}{1-\sigma(1+f)}\right)^{1/k_R}\right)} \tag{30}$$

Note that even with the signatures of an ideal length, this algorithm compares signatures for every pair of tuples in the cross product of R and S. If R and S each has one million tuples, there are one trillion comparisons.

4.4 Partition Based Algorithms

In this section, we propose a new algorithm for set containment joins over the nested internal representation that is based upon partitioning. In general, partition based algorithms for joins (scalar and spatial) attempt to optimize join execution by partitioning the problem into multiple smaller subproblems using a partitioning function. First, the relation R is partitioned into k partitions, $R_1, R_2, \ldots, R_k$. Similarly, the relation S is partitioned into $S_1, S_2, \ldots, S_k$ using the same function. Note that we are using a generalization of the classical definition of partitioning in that one tuple may be mapped to multiple partitions. An ideal partitioning function has the following characteristics:

- Each tuple r of relation R falls exactly in one of the partitions R_i $(1 \leq i \leq k)$
- Each tuple s of relation S falls exactly in one of the partitions S_i $(1 \leq i \leq k)$
- The join can be accomplished by joining only R_i with S_i $(1 \leq i \leq k)$

It is hard and expensive to satisfy the three conditions in non-scalar domains.

4.4.1 Will Partitioning Improve Speedup?

Partitioning algorithms have been shown to outperform other algorithms in scalar and spatial domains. In this section, we explore how partitioning speeds up set containment join. We analyze the speedup using a very simple analytical model. Let us assume that both the relations R and S have N tuples and that joins within the partitions are done using signature nested loops. (We relax this assumption later in this section). Also consider the class of partitioning functions in which only tuples of S are replicated. If r_S is the replication factor and P is the number of partitions, then the overall cost of the partition-based algorithm, ignoring constants, is given by

$$\sum_{P} \left(\frac{N}{P}\right) \times \left(\frac{r_S N}{P}\right) \tag{31}$$

On the other hand, the cost of the nested loop algorithm, again ignoring constants, is N^2. Hence the speedup of the partition-based algorithm is proportional to P/r_S, where r_S depends upon P.

To investigate this dependence, we make the assumption that the set instances draw their elements from the domain uniformly. Furthermore, we assume that the partitioning algorithm works by partitioning the elements of the domain from which the set is drawn. If a set has an element e, and e maps to a partition p, then the set itself must be mapped to partition p. From statistics [Fel57], assuming a large domain size (greater than set cardinality), the expected number of partitions to which a tuple is replicated (essentially r_S) when its set instance has cardinality of k is then given by

$$k - k(1 - 1/k)^P \tag{32}$$

According to equation (32), when the number of partitions is increased, the expected number of partitions approaches the value of k asymptotically (thereby bounding the replication to k). Hence the speedup of partition-based algorithm can be rewritten as

$$Speedup = \frac{P}{k - k(1 - 1/k)^P} \tag{33}$$

If k_S is the average set cardinality of relation S, the speed up can be easily rewritten as

$$Speedup = \frac{P}{k_S - k_S(1 - 1/k_S)^P} \tag{34}$$

A plot of equation (34) for various values of k_S is shown in Figure 21. The graph shows that as P increases the speedup increases. Consider the individual effects of the two terms in equation (34): P and $k_S - k_S(1 - 1/k_S)^P$. Increasing P tends to increase the speedup. However, increasing P also increases $k_S - k_S(1 - 1/k_S)^P$, which has the effect of decreasing the speedup. Since the rate of increase is greater than the rate of decrease of the denominator, overall the speedup increases. Replication is bounded as a tuple cannot be replicated to more partitions than its set cardinality. Once the replication has reached the maximum of k_S, the rate of increase is purely dominated by increase in P. Also, observe that for a given speedup to occur, higher cardinality sets require a larger number of partitions, since the decreasing effect of replication is extended in large sets. In order to counteract this effect, more partitions are required.

The number of partitions is bounded by the domain size. Hence the speed up is bounded by $\mid D \mid /k$. Of course, this analysis is greatly oversimplified and in practice such a speedup is not attainable, because increasing the number of partitions causes overhead of its own. However, the intuition from this simple model is valid: because there is a bound on the replication factor, increasing the number of partitions beyond a certain level will not cause any more replication.

Finally, we return to the assumption that each pair of partitions is joined using nested loops. Relaxing this assumption and considering faster algorithms for joining partitions

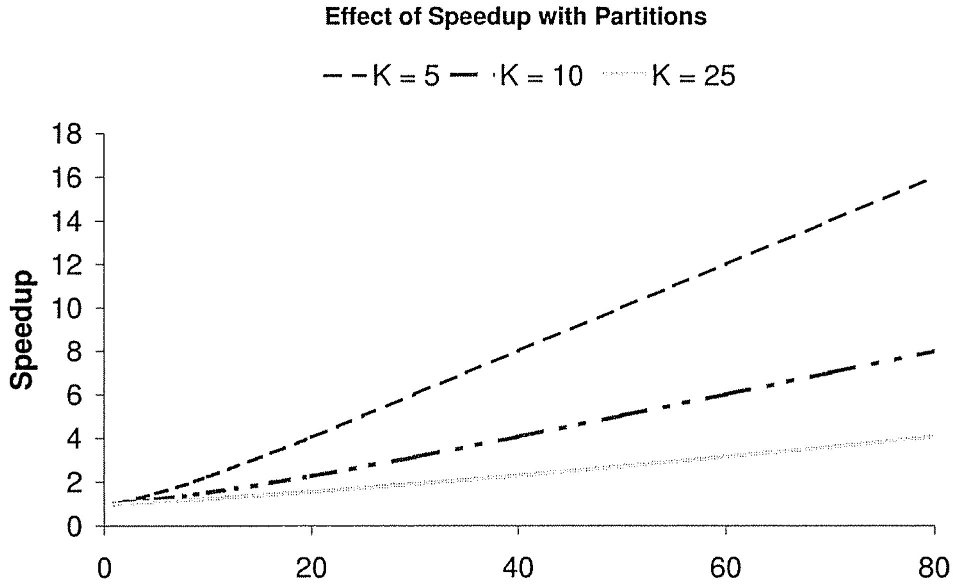


Figure 21: Theoretical Variation of Speedup with Increase in Partitions

does change the predicted speedup factors, but it does not affect the general conclusion of the model (that partitioning is beneficial).

4.4.2 Partitioned Set Join Algorithm (PSJ)

Now we are ready to describe the Partitioned Set Join Algorithm (PSJ), which uses a two level partitioning scheme. It operates in three phases:

- **Partitioning Phase**: Each tuple of R is sent to exactly one partition based on the first level partitioning function h. Each tuple of S, in general, is replicated across multiple partitions using (the same) h.

- **Joining Phase**: Each partition of R is joined with its counterpart in S using a second level partitioning function that operates on signatures. Hence false drops are possible.

- **Verification Phase**: The tuple pairs that the join phase indicates could join, are compared to remove any false drops.

The subsequent sections describe each of the phases in detail.

4.4.3 Partitioning Phase

This phase uses a partitioning function h that operates on the set elements. The partitioning phase begins by reading the relation R. For each tuple r of R, the following steps are executed

1. A 3-tuple (c_i, s_i, OID_i) is computed, where c_i is the set cardinality, s_i is the signature of the set instance, and OID_i is the OID of the tuple.

2. A random element e_R is picked from $r.\{b\}$.

3. The 3-tuple is sent to the partition determined by $h(e_R)$.

Observe that the 3-tuple for each tuple of R is sent only to one partition. Now the relation S is read. For each tuple s of S, the following steps are executed

1. A 3-tuple (c_i, s_i, OID_i) is computed.

2. *For each* element $e_S \in s.\{d\}$, the 3-tuple is sent to the partition determined by $h(e_S)$.

Note that if $r.\{b\} \subseteq s.\{d\}$, then the partition determined by $h(e_R)$ will contain the 3-tuples corresponding to r and s. Hence the algorithm computes containment correctly.

4.4.4 Joining Phase

During the joining phase, each partition of R is joined with its counterpart in S. There are various algorithms that could be used in this phase. However, at this point, the tuples in each partition do not carry the actual set instances, since they are approximated by signatures. Hence the join algorithm in this phase has to operate directly on signatures. In this phase, we use a partition based in-memory algorithm using signatures.

The joining algorithm works in two steps: the *build step* and the *probe step*. In the build step, an array A of size equal to the number of bits in the signature is constructed. Now the partition R_i is scanned and each 3-tuple (c_i, s_i, OID_i) is read. A bit position m that is set to 1 is chosen randomly from the signature. The 3-tuple is inserted into $A[m]$. At the end of first step, the signatures from partition R_i have been partitioned.

During the probe step, partition S_i is scanned. For each 3-tuple (c_j, s_j, OID_j), the chain of signatures in $A[n]$ is examined whenever bit n is set to 1 in s_j. The containment predicate is evaluated (as in Section 4.3.2) for each signature encountered in the chain and the candidate pairs (OID_i, OID_j) are inserted into a temporary relation. These candidate pairs potentially satisfy the containment relationship.

This phase of the algorithm is similar to signature hash join (SHJ) proposed in [HM97]. We use a single bit in the signature to determine the array index for R. SHJ in general uses more bits (a partial signature) to determine the array index. For S, SHJ requires all possible subset signatures to be enumerated for a given partial signature to determine the chains to be probed. This enumeration is exponential in the size of the signature.

4.4.5 Verification Phase

In the verification phase, we examine the actual R and S tuples to determine whether they satisfy the join condition. The main issues involved in this phase are speeding up set containment verification and avoiding random seeks while fetching the tuples.

The set elements in the tuples of R and S retrieved from the storage manager are stored in the nested internal representation (as described in Chapter 2). Such a storage representation is not very efficient for evaluating the containment predicate for the OID pair (OID_{iR}, OID_{jS}). This is because it requires examining all the set elements of the tuple corresponding to OID_{iS} for every set element of the tuple corresponding to OID_{iR}.

In order to speed up the containment verification, the set elements of each tuple of R are inserted into a hash table to facilitate faster probing. Observe that each tuple has a hash table of its own. The set instances of each tuple in S can either be directly scanned in the nested internal representation sequentially or can be converted into an in-memory array representation and accessed. The former approach is expensive since each set element has to be converted into an in-memory representation before probing into the hashed set instances of R. Hence it is not efficient if the same tuple is accessed repeatedly. On the other hand, the array representation is advantageous since it amortizes the cost of conversion into an array over multiple accesses, thereby improving the overall time. Using these combinations, we get the best speedup for set containment checking.

In order to minimize the disk seeks we employ a strategy described in [Val87]. The OID pairs relation is already in sorted order with OID_{iR} as the primary key. This makes the access to R tuples sequential. Now as many S tuples as possible are fetched such that the available memory holds (i) tuples of R and its hashed set instances, and (ii) tuples of S and its set instances in array form and (iii) the corresponding array of

(OID_{iR}, OID_{jS}) pairs. The OIDs of R are swizzled to point to the R tuples in memory and then the array is sorted on OID_{jS} so that the access to the tuples of S is sequential. The nested set instances present in the swizzled tuples of R are converted into efficient hashed representations in memory. Then the S tuples are read sequentially into memory and their nested set-valued attribute is converted into an array representation. The join condition is evaluated for each OID pair by scanning the set instances in S in array representation and probing into the hashed instances of R till all set elements of R are accounted for. We chose to build a hash table for set instances of R rather than S since they are smaller, reducing the cost of building the hash table.

4.4.6 Estimation of the Number of Partitions

As seen in Section 4.4.1, the number of partitions has a critical impact on the performance of PSJ. The desirable number of partitions depends on two parameters: the average set cardinality, and the relation cardinality of the relations involved. Even though the speedup is expected to increase as the number of partitions is increased, in practice, the overhead associated with each partition prevents such unbounded speedup.

In order to estimate the desired number of partitions (P_{PSJ}) and the signature size (F_{PSJ}), we employed a detailed analytical model to predict the execution time of the algorithm. Using the model, the cost of each phase is calculated.

Cost of Partitioning Phase

The cost of partitioning (C_{PP}) includes the cost of reading the relations, cost of determining the partitions and the partitioning overhead. It can be calculated as

$$C_{PP} = \begin{cases} \| R \| \, IO_{seq} + \| S \| \, IO_{seq} + & \text{I/O cost of reading the relations } R \text{ and } S \\ P_{PSJ} \left\lceil \frac{|R| \times T}{P_S \times P_{PSJ}} \right\rceil IO_{rand} + & \text{I/O cost of writing the partitions of } R \\ P_{PSJ} \left\lceil \frac{r_S |S| \times T}{P_S \times P_{PSJ}} \right\rceil IO_{rand} + & \text{I/O cost of writing the partitions of } S \\ h_c \mid R \mid + h_c k_S \mid S \mid & \text{CPU cost of computing the hash function} \end{cases} \tag{35}$$

where T is the size of 3-tuples. It is equal to $(TID_s + \lceil F_{PSJ}/8 \rceil + i)$ where i is the number of bytes requires to store the set cardinality. The cost of partitioning assumes that at least a page of each partition fits in memory.

Cost of Joining Phase

The cost of the joining phase (C_{JP}) is a summation of the joining cost of the individual partitions. The joining cost in turn depends on the cost of reading the partitions, the number of signature comparisons, and writing the result. Estimating the total number of signature comparisons depends on (i) length of the chain and (ii) expected number of chains examined for each signature of a partition of S. Assuming a uniform distribution, one can determine the length of the chain C_L as

$$C_L = \frac{\mid R \mid}{F_{PSJ} P_{PSJ}} \tag{36}$$

Now we have to determine the expected number of chains examined for a signature from a tuple of S. It depends on the number of bits set to 1. The probability that a bit position b is set to 1 is given by $1/F_{PSJ}$. If m is the expected number of bits set to 1, then we have

$$m = F_{PSJ} \times Prob\{b_t^i = 1\} \tag{37}$$

Now m can be rewritten as

$$m = F_{PSJ}\left(1-\left(1-\frac{1}{F_{PSJ}}\right)^{k_S}\right) \tag{38}$$

Since m also equals the number of chains examined, the total number of comparisons N_p required for each partition is given by

$$N_p = \frac{m \mid R \mid\mid S \mid}{F_{PSJ}P_{PSJ}^2} \tag{39}$$

Now the total cost of joining can be calculated as

$$C_{JP} = \begin{cases} P_{PSJ}\left\lceil \frac{|R|\times T}{P_S \times P_{PSJ}} \right\rceil IO_{seq}+ & \text{I/O cost of reading the partitions of } R \\ P_{PSJ}\left\lceil \frac{r_S|S|\times T}{P_S \times P_{PSJ}} \right\rceil IO_{seq}+ & \text{I/O cost of reading the partitions of } S \\ \frac{m|R||S|s_c}{F_{PSJ}P_{PSJ}}+ & \text{CPU cost of comparing the signatures} \\ \left\lceil \frac{\sigma|R||S|V}{P_S} \right\rceil IO_{seq} & \text{I/O cost of writing the result} \end{cases} \tag{40}$$

where V is the size of the OID pairs which is $2 \times TID_s$.

Cost of Verification Phase

The cost of verification (C_{VP}) includes the I/O cost of fetching the tuples and CPU cost of evaluating the containment. It is calculated as

$$C_{VP} = \begin{cases} 2\left\lceil \frac{\sigma|R||S|V}{P_S} \right\rceil IO_{seq}+ & \text{I/O cost of sorting the result} \\ \sigma \mid R \mid\mid S \mid e_c log(\frac{MP_S}{V})+ & \text{CPU cost of sorting the result} \\ 2\sigma \mid R \mid\mid S \mid IO_{seq}+ & \text{I/O cost of fetching the tuples in the worst case} \\ \sigma \mid R \mid\mid S \mid h_c(k_R + k_S) & \text{CPU cost of verifying the set containment} \end{cases} \tag{41}$$

The overall cost of the algorithm is the sum of the cost of these individual phases. The cost of the algorithm is minimum when the number of partitions is optimal. In order to derive an equation for the partitions, we differentiate the total cost with respect to P_{PSJ} and equate to zero and solve for P_{PSJ}. However, if we differentiate directly the overhead for fragmentation in the partitions will not be taken into account. This is because of the presence of the ceiling functions in various phases. Fragmentation and other partitioning overhead heavily depends on the implementation of the system. They depend on the size of the page, the overhead per tuple, the time to create and delete partitions, the cost of pinning and un-pinning a page in the partition.

In order to model the partitioning overhead for our system, we tried a few experiments. Based on these observations we modeled the overhead as follows:

- Approximate the fragmentation effect as a quadratic function in P_{PSJ} as $0.5P_{PSJ}^2 IO_{rand}$ in the partitioning phase for each relation and $0.5P_{PSJ}^2 IO_{seq}$ for reading the partitions again in the joining phase.

- The overhead of creating and destroying partitions is again approximated by a quadratic function in P_{PSJ} and modeled by a term of the form HP_{PSJ}^2 where H is a system dependent constant.

Even though the replication factor of S relation is related to P_{PSJ}, as described Section 4.4.1, it is substituted by its average set cardinality for simplification. After these additions and substitution for m, the overall cost of the algorithm is differentiated with respect to P_{PSJ}. We get

$dC/dP_{PSJ} =$

$$2P_{PSJ}(IO_{rand} + IO_{seq} + H) - \mid R \mid\mid S \mid \left(1 - \left(1 - \frac{1}{F_{PSJ}}\right)^{ks}\right) s_c / P_{PSJ}^2$$

Setting $dC/dP_{PSJ} = 0$ and solving for P_{PSJ} gives

$$P_{PSJ} = \left(\frac{\mid R \mid\mid S \mid \left(1 - \left(1 - \frac{1}{F}\right)^{ks}\right)}{Z} \right)^{1/3} \qquad (42)$$

where $Z = 2IO_{rand} + 2IO_{seq} + H$

The fudge factor H accounts for various system dependent factors including the cost of creating and destroying partitions and other overheads. The fudge factor is likely to vary across systems. For a given system, H can be determined as follows: for a set of sample data, run PSJ for various number of partitions and measure the partition creation and deletion times. For each of the time, divide by P^2_{PSJ} for various number of partitions and calculate the average value of H.

If the number of buffer pool pages available is less than the estimated value, whatever is available is committed to the algorithm.

4.4.7 Estimation of Signature Size

The performance of PSJ depends on the size of the signature used for approximating sets. Since partitioning avoids many redundant comparisons, one can expect the signature size (F_{PSJ}) to be lower (when compared to Sig-NL). Also, as the number of partitions is increased, the signature size is expected to get lower. We derive an equation for approximately estimating the signature size based on desirable number of false drops. In order to compute the total number of false drops, first the number of false drops per partition has to be determined. The false drop probability equation (26) can be rewritten as

$$P_{FD_p} = \frac{falsedrops_p}{N_p - resultsize_p - falsedrops_p} \tag{43}$$

where P_{FD_p} is the false drop probability per partition, $falsedrops_p$ is the number of false drops per partition, $resultsize_p$ is the size of the result after joining corresponding partitions of R and S and N_p is the total number of signature pair comparisons for joining the partition. N_p is given by equation (37).

Expressing the false drops f_p as a percentage of the result size of the partition and substituting for m, the equation (43) can be rewritten as

$$P_{FD_p} = \frac{f_p \sigma_p \mid R \mid r_S \mid S \mid}{\mid R \mid\mid S \mid \left(1 - \left(1 - \frac{1}{F_{PSJ}}\right)^{r_S}\right) - \sigma_p \mid R \mid r_S \mid S \mid - f_p \sigma_p \mid R \mid k_S \mid S \mid} \tag{44}$$

where σ_p is the selectivity of the join per partition. Assuming a uniform distribution, the partition selectivity can be computed by observing that the total result is the summation of the results from individual partitions.

$$\sigma \mid R \mid\mid S \mid = P_{PSJ} \mid R \mid k_S \mid S \mid \sigma_p / P_{PSJ}^2 \tag{45}$$

Rearranging, we get

$$\sigma_p = \sigma P_{PSJ} / r_S \tag{46}$$

Because of uniform distribution we can further assume that $f_p = f$ and $P_{FD_p} = P_{FD}$ where f is the overall percentage of false drops and P_{FD} is the overall false drop probability. Now the equation can be rewritten as

$$P_{FD} = \frac{f \sigma P}{\left(1 - \left(1 - \frac{1}{F_{PSJ}}\right)^{k_S}\right) - \sigma P - f \sigma P} \tag{47}$$

Combining equations (29) and (47) we get

$$(1 - e^{-k_S/F_{PSJ}})^{k_R} - \frac{f \sigma P}{\left(1 - \left(1 - \frac{1}{F_{PSJ}}\right)^{k_S}\right) - \sigma P - f \sigma P} = 0 \tag{48}$$

We use the bisection method to solve this equation. There is a cyclic dependency between equations (42) and (48) Hence both the equations have to be solved simultaneously. We use these equations to determine the appropriate combination of partitions and signature size in our experiments for PSJ. As we shall see in Chapter 5 (Section 5.5 and Section 5.6), fortunately the performance curves as a function of the number of partitions and signature size are rather flat. So these equations do not have to be exact to obtain reasonable performance.

4.5 Summary

This chapter investigated algorithms for computing a set containment join. These algorithms cover two possible implementations of set-valued attributes: the unnested external representation, and the nested internal representation. The unnested external representation is used by commercial O/R DBMSs for implementing set-valued attributes. In this case, the set containment join is implemented using a standard SQL2 query. For the nested internal representation, this chapter considers two algorithms. The first is a variation of nested loops (Sig-NL) that uses signatures to speed up the evaluation of the join predicate. The second algorithm is PSJ, a new partition based algorithm that is proposed in this chapter. This algorithm is based on a two-level partitioning scheme by using set elements to partition relation R and replicate relation S. Within each partition, it uses an in-memory algorithm based on partitioning of signatures.

Chapter 5

Performance Evaluation of Set Containment Join Algorithms

5.1 Introduction

In this chapter, we evaluate the performance of the three set containment algorithms: the SQL approach for the unnested external representation (**SQL**), and the signature nested-loops (**Sig-NL**) and **PSJ** algorithms for nested internal representations, by implementing them in an object relational system. As a special case, we also ran PSJ with one partition, which we call **PSJ-1**. The special case of one partition is important when applicable, because it has no partitioning overhead. We first describe our implementation of these algorithms and then present results from various experiments designed to investigate the performance of these algorithms under various conditions.

5.1.1 Chapter Organization

The rest of the chapter is organized as follows. Section 5.2 describes the implementation of various algorithms in an O/R DBMS. The set distributions we used in our experiments are described in Section 5.2.2. The effect of varying relation cardinalities is described in Section 5.3. An investigation of the SQL approach is presented in 5.3.1. Section 5.4 studies the effect of varying set cardinality. Sections 5.5 and 5.6 describe in detail the impact of signature size and number of partitions on the performance of PSJ. Intermediate disk space requirements for PSJ are shown in Section 5.7. Finally, Section 5.8 summarizes the results of the performance evaluation.

5.2 Implementation

Paradise is a shared nothing parallel object-relational system developed at the University of Wisconsin-Madison [PYK+97]. We implemented sets using the ADT mechanism in Paradise. As mentioned in Chapter 3, the set ADT implements a number of set-oriented methods, including: create-iterator, which returns an iterator over the elements of the set; and set operators which are implemented by type specific methods invoked by the query engine when comparison and assignment are performed on sets.

We implemented signature-nested loops (Sig-NL) and PSJ as join algorithms in the system, and extended the optimizer to recognize set containment join operations in queries so that it can schedule the appropriate operator for execution. For the SQL approach, the magic set optimization was used to rewrite the correlated nested query as shown in Section 4.3.1. In order to ensure that the optimizer did not choose bad plans, optimal physical plans for each query were fed into the system rather than the queries themselves. The plans are shown in Figure 22.

5.2.1 Experimental Setup and Data Generation

In our experiments, the total size of the non set-valued attributes in a tuple was 68 bytes. The average size of each set element was 30 bytes. We ran the experiments on an Intel 333 MHZ Pentium processor with 128MB of main memory running Solaris 2.6. We used a 4GB disk for storing the database volume. The disk was mounted as a raw

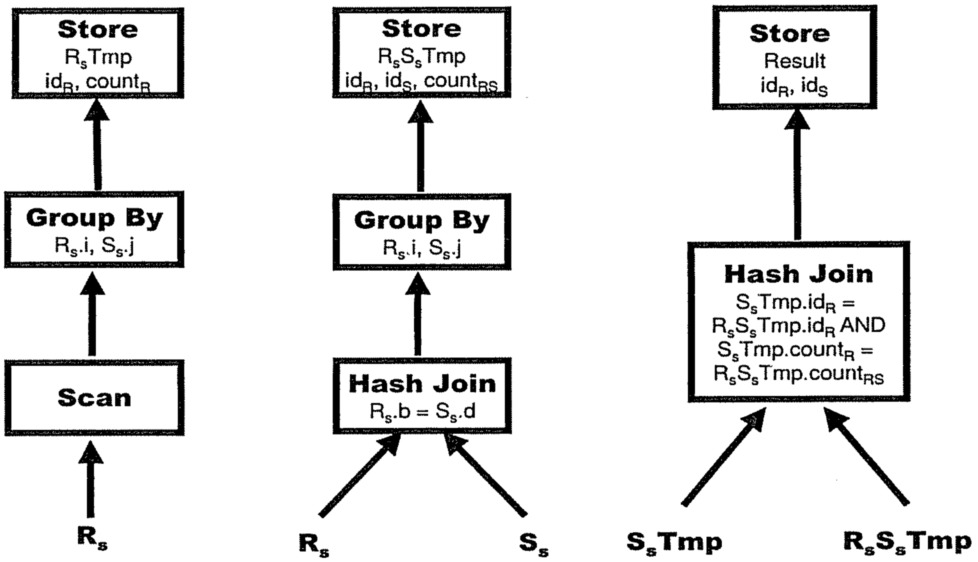


Figure 22: Physical Plans for Count, Candidate and Verify Queries Respectively

device. It provided an I/O bandwidth of 6 MB/sec. Paradise was configured with a 32MB buffer pool. Though this buffer pool size may seem small compared to current trends in memory, we used this value since we wanted to test data sets that were much larger than the buffer pool. As will be seen in the following sections, with this buffer pool size, some experiments take many days to run. Each experiment was run against a cold buffer pool to eliminate the effect of caching.

The data generator for the BUCKY benchmark [CDN$^+$97] was modified to generate data for our experiments. The data generator takes as input the cardinality of the relations R and S, the average cardinality of the set-valued attributes in the two relations, the size of the domain from which the set elements are drawn, and a correlation value. For each tuple, the set-valued attribute is generated as follows. First, the data generator divides the entire domain into 50 smaller sub-domains. The set elements are drawn from these sub-domains. Set elements are correlated if they are drawn from the same sub-domain. Correlation of a set instance is defined as the percentage of the set elements that are drawn from a single sub-domain. For example, if the set cardinality is 10, a correlation of 90% implies that 9 set elements are picked from one sub-domain and 1 element is randomly chosen from one of the remaining 49 sub-domains. All the experiments used a correlation of 10% unless otherwise specified. Joining tuples were generated such that every R tuple joins with exactly one S tuple. Finally, we chose the response time as our performance metric.

5.2.2 Set Distributions

There are many distributions involving set-valued attributes because there are many degrees of freedom:

- average set cardinality of relation R and S,
- relation cardinality of R and S,
- size of domain from which the set elements are drawn, and
- the degree of correlation among the elements.

Each parameter can influence the performance of the containment join algorithm. In an effort to reduce the problem space, we restricted ourselves to varying the relation and set cardinalities. Based on these two parameters, we have four possible quadrants, as shown in Figure 23.

Here we give an example for each quadrant. If sets are mainly used as a logical collection (e.g, a set of courses, a set of pre-requisites, a set of hobbies, a set of outgoing links in a web page) the average set cardinality is likely to be small (typically in the range

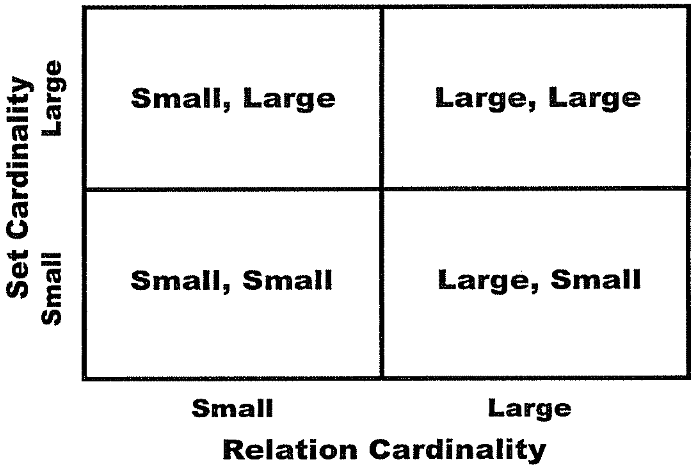


Figure 23: Taxonomy of Set Distributions

of 5-10) while the relation cardinality might be potentially large (number of students, number of employees). On the other hand, if each set instance is considered as a relation (e.g., the set of employees working for a department), then the average set cardinality can be large, however the relation cardinality might be small (number of department in a company). In the XML world since anything can be represented as sets, it is possible that the average set cardinality and relation cardinality potentially become large (e.g., the number of documents).

5.3 Experiment 1: Varying Relation Cardinality

In this set of experiments, we investigated the effect of varying the relation cardinality. The domain size was fixed at 10,000. Since the containment join was not symmetric, we further refined the experiments based on different cardinalities of R and S :

- The relation cardinalities of R and S were varied together
- The relation cardinality of S was kept constant at a large value and that of R was varied.

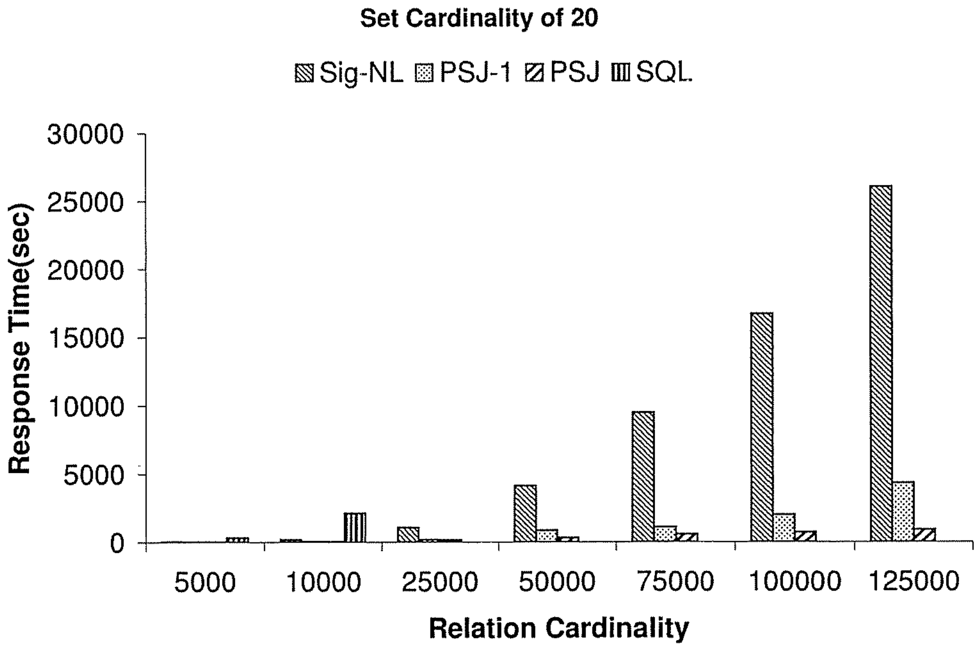


Figure 24: Varying Relation Cardinality for Set Cardinality of 20

- The relation cardinality of R was kept constant at a large value and that of S was varied.

Varying Relation Cardinalities of R and S

In this experiment, the relation cardinality was varied for two values of set cardinality: 20 and 120. The results of these experiments are plotted in Figure 24 and Figure 25. The numbers for the SQL approach for relation cardinalities greater than 20,000 are not included in the figure since these runs took more than 24 hours. The main observation is that PSJ outperforms (or performs as well as) the other algorithms consistently over the entire space of relation cardinality. On the other hand, the SQL approach starts getting worse from 10,000 tuples onwards. Section 5.3.1 discusses why the SQL approach performs poorly. Sig-NL and PSJ are analyzed in Section 5.3.2.

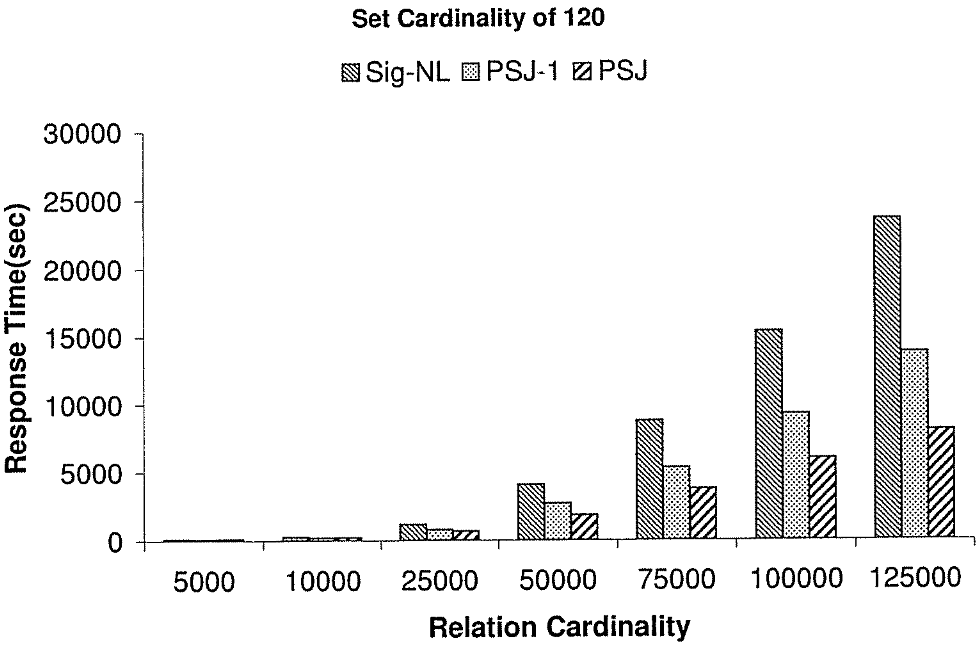


Figure 25: Varying Relation Cardinality for a Set Cardinality of 120

5.3.1 Performance of the SQL Approach

As may be seen from Figure 24, the SQL approach performs reasonably well at very small relation and set cardinalities. However, as the relation sizes increase (note the peak at 10,000), the response time increases rapidly. The cost breakdown of the SQL approach is shown in Figure 26. The figure shows the times taken for running each of the component queries. It is evident that most of the time is dominated by the candidate generation query. The candidate generation query is expensive for the following reasons:

- The input to the joins are two large set relations R_S and S_S. These relations suffer from cardinality explosion (their cardinality is the product of average set cardinality and relation size of the base relations). Such an explosion makes the join expensive.

- The number of intermediate tuples that are generated as a result of the join is also large. The output of the join generates a tuple for every element in R_S and its intersecting set in S_S. Essentially, it is computing an intersection which is a

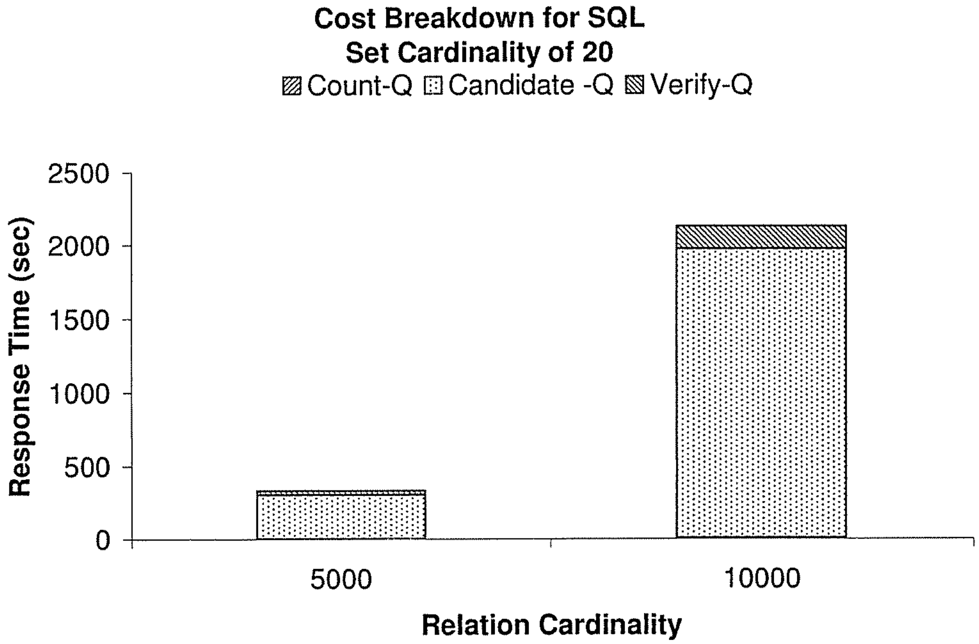


Figure 26: Cost Breakdown for SQL Approach

superset of the actual result. Note that the probing phase of the join and the aggregate phase were run in a pipelined fashion so that there is no intermediate I/O cost.

- The number of groups generated from the aggregate operator is also large. The number of groups is proportional to $| R | \times | S |$ and it is equal to the number of set pairs that have at least one element in common. The number of intermediate tuples (groups) actually generated is plotted against relation cardinality, in Figure 27. The figure confirms the enormity of the number of tuples (groups) generated. This number is large even for smaller set and relation cardinalities.

Because of the aforementioned problems and consequent performance degradation, the SQL approach is not considered in the remaining sections.

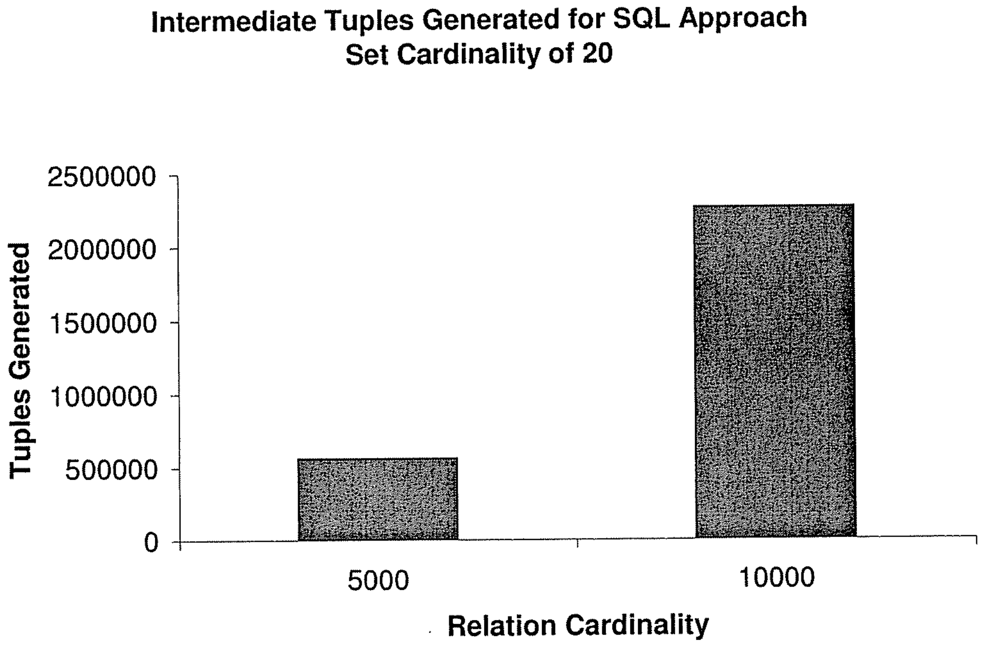


Figure 27: Intermediate Tuples Generated

5.3.2 Sig-NL vs PSJ

The individual cost breakdown of Sig-NL is shown in Figure 28 and Figure 29 for a set cardinality of 20 and 120 respectively. Similarly for PSJ-1 and PSJ they are illustrated in Figures 30, 31, 32 and 33.

In general, the cost of these algorithms consists of three components: the partitioning cost, the comparison cost and the verification cost. Sig-NL and PSJ-1 do not incur any partitioning cost. The comparison cost is high in Sig-NL. It decreases in PSJ-1 and is least in PSJ.

The first observation is that PSJ outperforms PSJ-1 and Sig-NL consistently as may be seen from Figure 24 and Figure 25. The basic insight is that if PSJ is to perform well, the reduction in the number of comparisons should be significant and the partitioning cost should not be too high. The reduction in number of comparisons is dominant at higher relation cardinalities, as may be seen in Figure 30 and Figure 32. Hence PSJ consistently

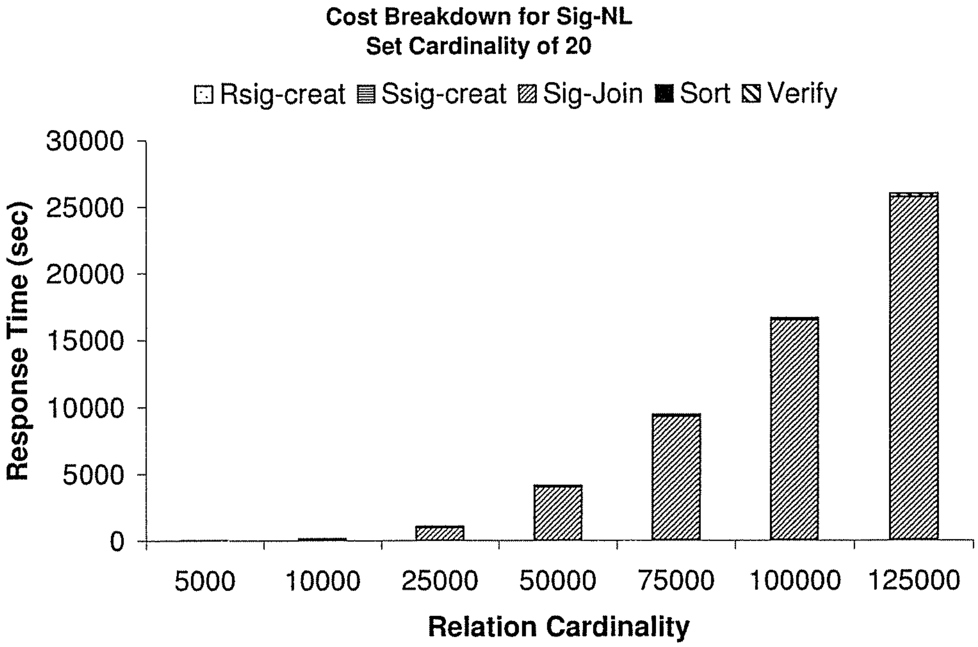


Figure 28: Cost Breakdown of Sig-NL for a Set Cardinality of 20

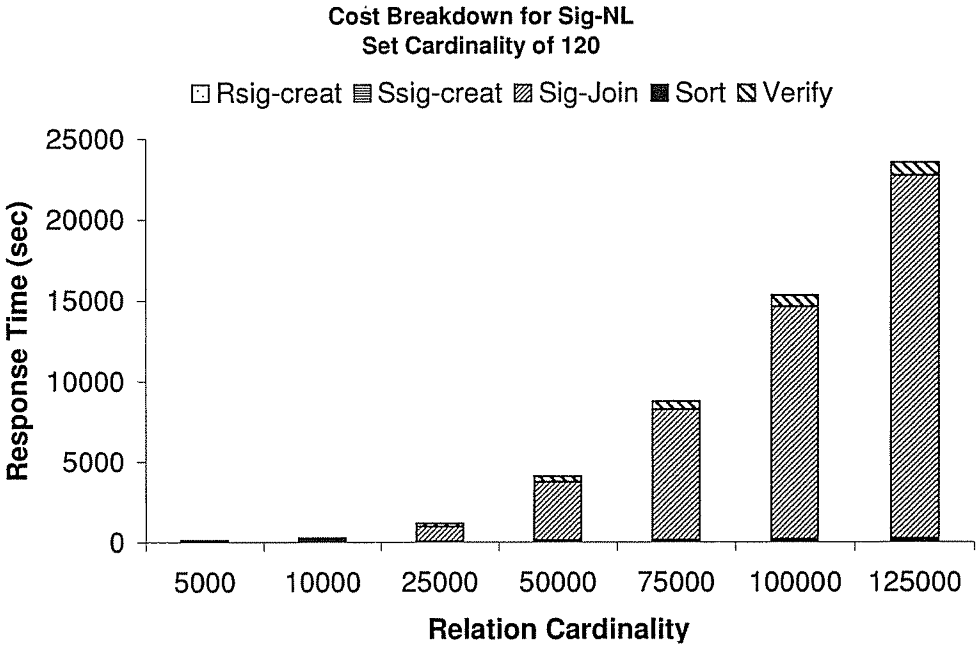


Figure 29: Cost Breakdown of Sig-NL for a Set Cardinality of 120

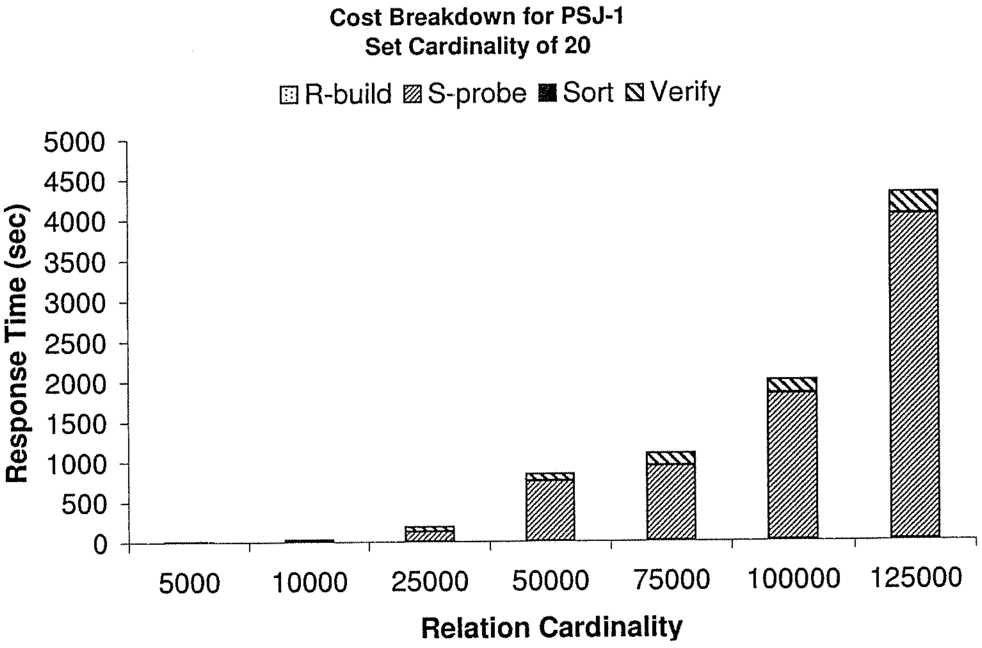


Figure 30: Cost Breakdown of PSJ-1 for a Set Cardinality of 20

performs better at higher relation cardinalities. For lower relation cardinalities, the cost gained by avoiding unnecessary comparisons is not high. The gap between PSJ and the rest is smaller for set cardinality of 120. This is because the partitioning cost is higher. In addition, the comparison cost also increases because of replication.

Another contributing factor is the requirement of large signature sizes for lower set cardinalities of R in Sig-NL and PSJ. This might seem counterintuitive. However, a closer look at the false drop probability equation (29) reveals the following characteristics:

- For a given number of false drops and for a constant set cardinality of S, as the average set cardinality of R increases, the size of the signature decreases. For a given signature size, when the set cardinality increases, more bits get turned to 1 in R tuple signature. Hence the probability of a false drop is now reduced since for a tuple to match this signature it must match with the signature in all these bit positions in S tuple signature. As the false drops are kept constant, the effect of

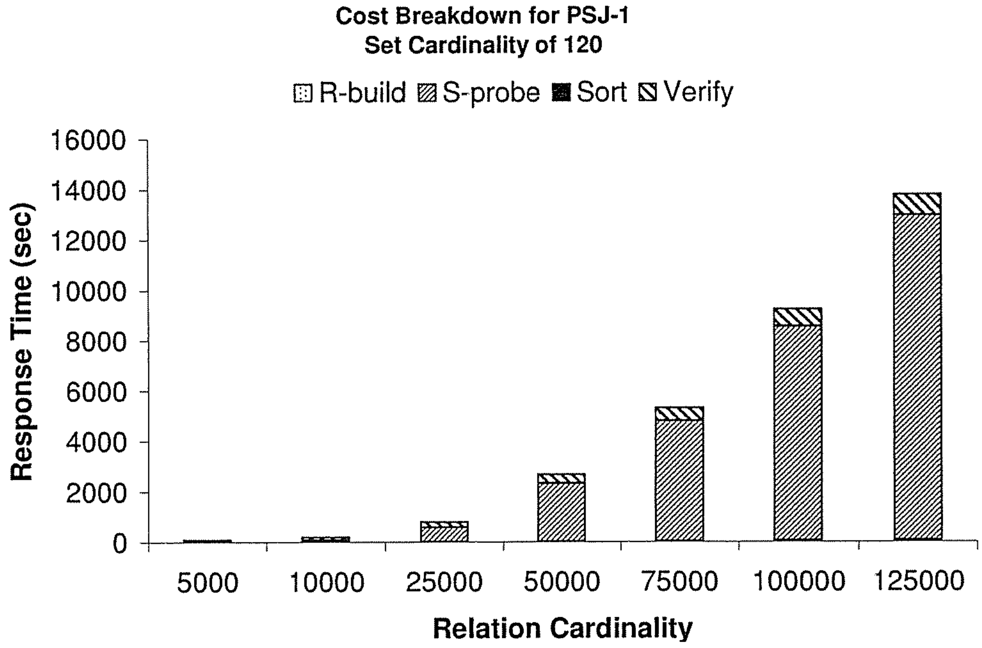


Figure 31: Cost Breakdown of PSJ-1 for a Set Cardinality of 120

increasing cardinality decreases the signature size.

- For a given number of false drops and for a constant set cardinality of R, as the average set cardinality of S increases, the size of the signature increases. As more bits are turned to 1 in a S tuple signature, the probability of a false drop is increased. Because it highly likely that bits turned to 1 in R tuple signature will also be turned to 1 in S tuple signature. Since the false drops are kept constant, the effect degenerates into increase in signature size.

Because of the above two opposing effects, an increase in set cardinality decreases the signature size initially reaching a minimum before starts increasing again. Hence in order to keep the false drops to a minimum, an increase in the signature size is required for smaller set cardinalities. For example, in Sig-NL when the relation cardinality of R (and S) was 25000, the required signature size was 181 bits for a set cardinality of 20 while it was 104 bits for a set cardinality of 120. Note however that as the average set cardinality

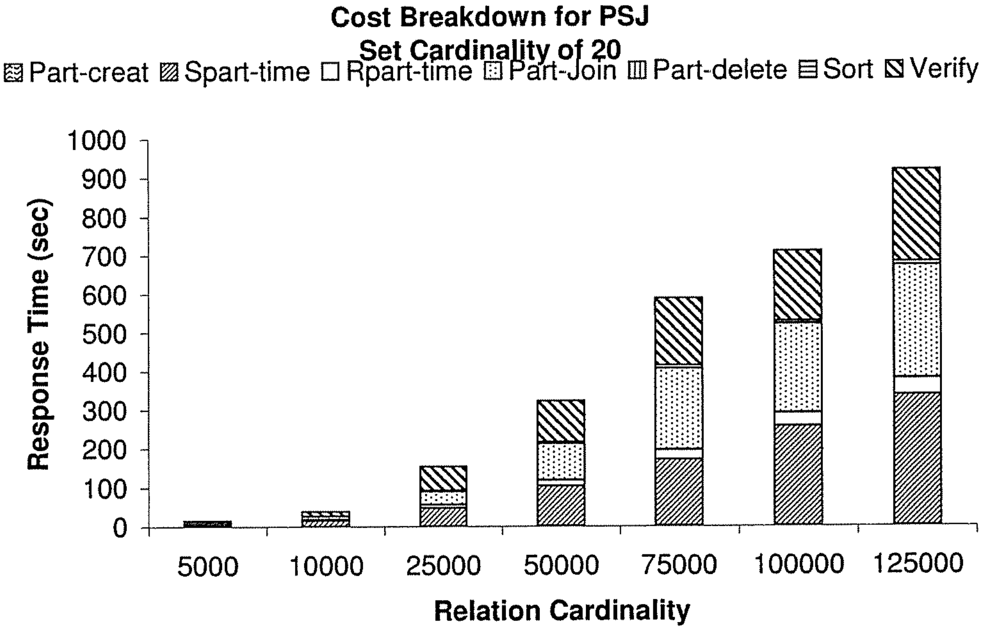


Figure 32: Cost Breakdown of PSJ for a Set Cardinality of 20

of S increases, the signature size increases as expected.

The second observation is that PSJ-1 outperforms Sig-NL consistently. This is expected since several unnecessary comparisons are eliminated. Quantitatively, for a set cardinality of 20 and relation cardinality of 25000, Sig-NL requires 625 million comparisons whereas PSJ-1 requires only 80 million comparisons. When the set cardinality is 120, the number of comparisons increases since the expected number of bits set to 1 in the signature increases thereby causing more chains to be examined for a given set of S. Hence the performance gap between the two decreases.

We also conducted experiments where the cardinality of one relation was fixed and the other was varied. The trends observed were the same.

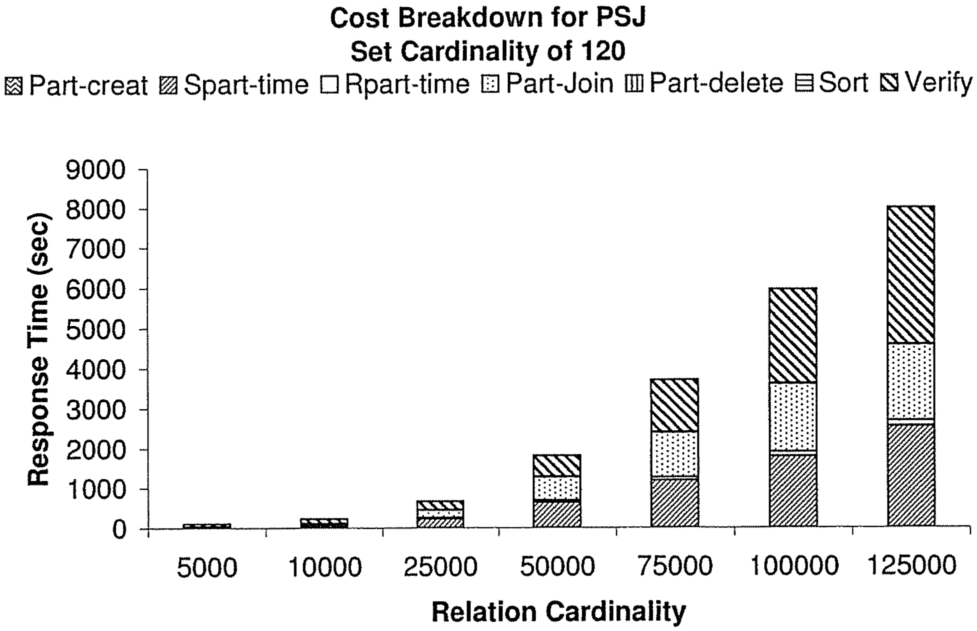


Figure 33: Cost Breakdown of PSJ for a Set Cardinality of 120

5.4 Experiment 2: Varying Set Cardinality

In this experiment, we varied the set cardinality for two different relation cardinalities, 20,000 and 100,000 to explore the quadrants of small and large relation cardinalities. The signature size for Sig-NL and PSJ-1 and the number of partitions for PSJ were chosen using equations (42) and (48). The domain size was set at 10,000. The results are plotted in Figure 34 and Figure 35 for relation cardinalities of 20,000 and 100,000. The individual cost breakdown of Sig-NL is shown in Figure 36 for a relation cardinality of 20,000 and Figure 37 for relation cardinality of 100,000. For PSJ-1, individual costs are shown in Figure 38 for a relation cardinality of 20,000 and Figure 39 for relation cardinality of 100,000. Figure 40 and Figure 41 shows the same for PSJ.

For a given relation cardinality, as the set cardinality increases, the gap between PSJ and the rest diminishes. In fact, for a relation cardinality of 20,000, when the set cardinality is 160, PSJ-1 outperforms PSJ. This is because the partitioning cost increases

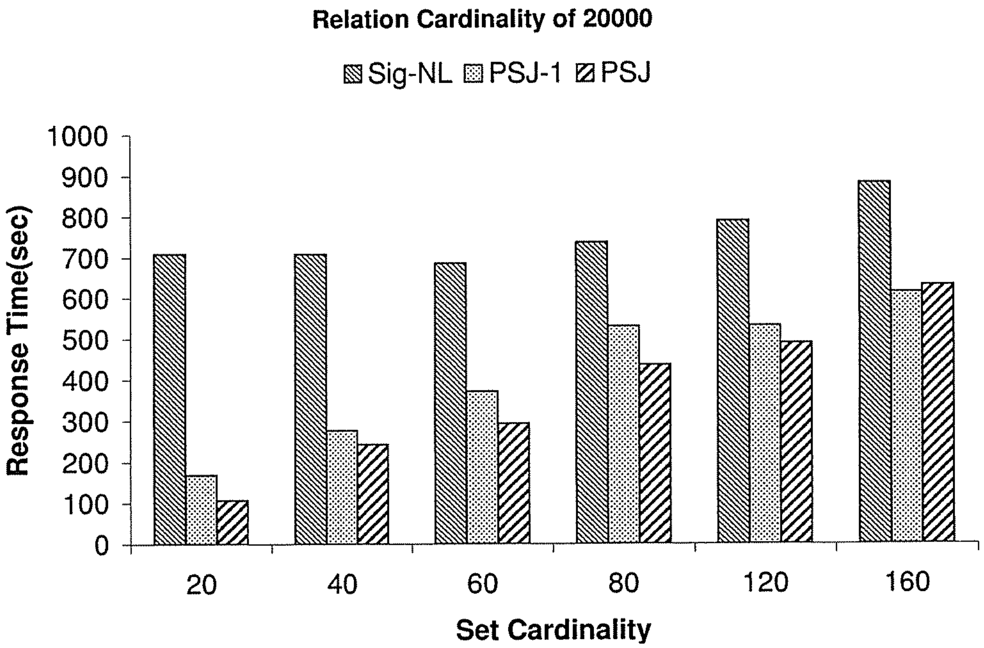


Figure 34: Varying Set Cardinality for Relation Cardinality of 20,000

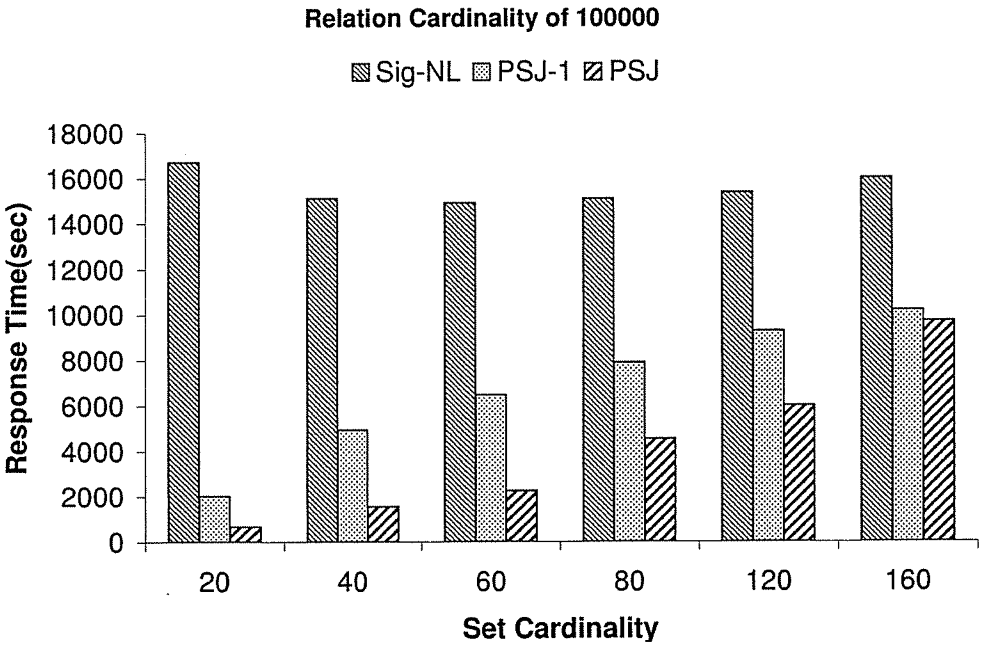


Figure 35: Varying Set Cardinality for Relation Cardinality of 100,000

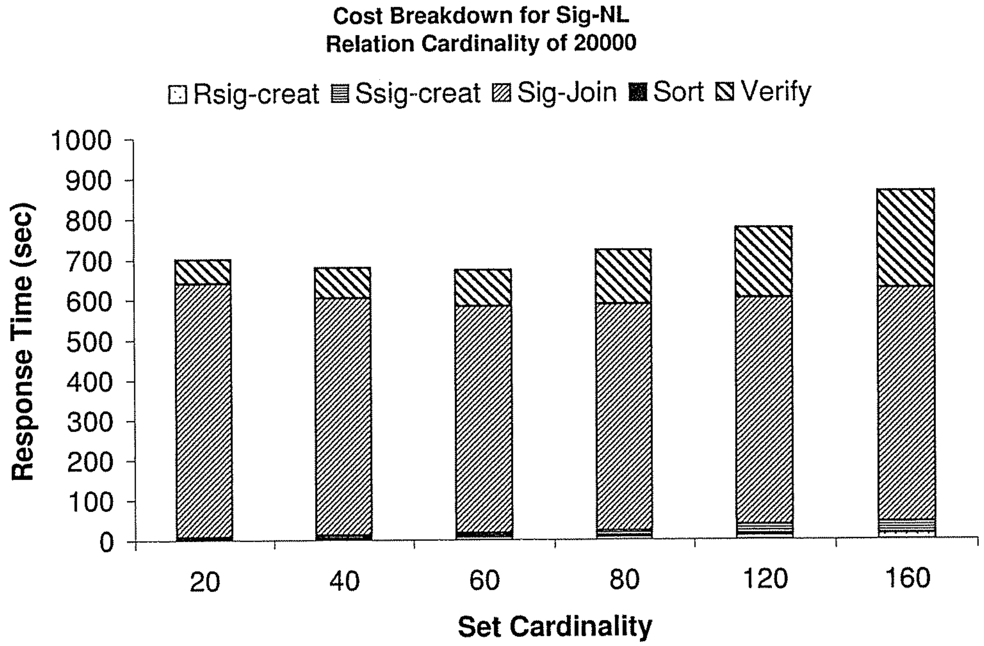


Figure 36: Cost Break Down of Sig-NL for Relation Cardinality of 20,000

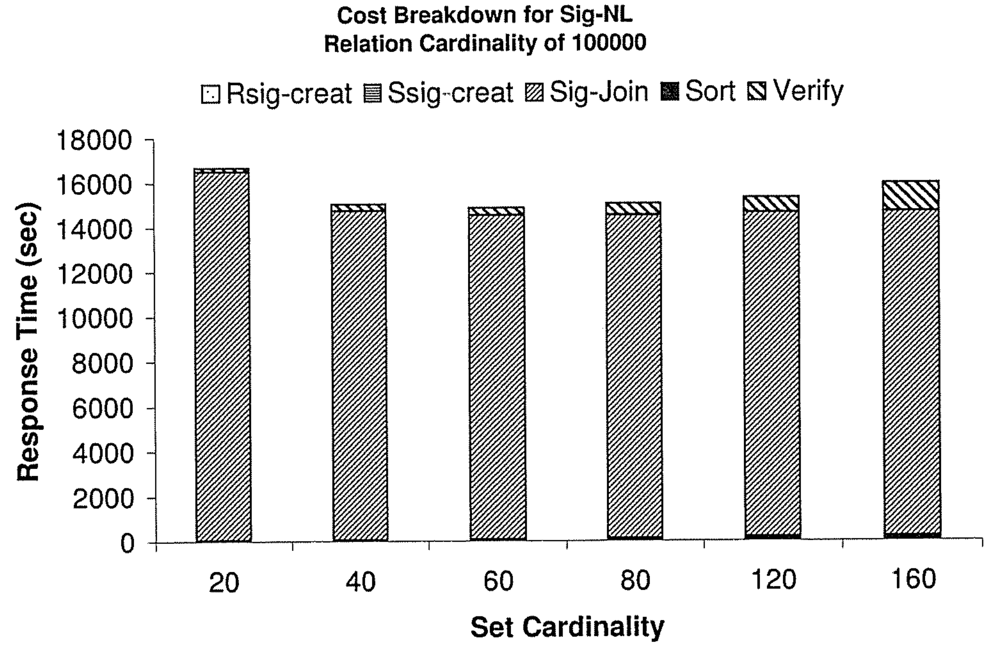


Figure 37: Cost Break Down of Sig-NL for Relation Cardinality of 100,000

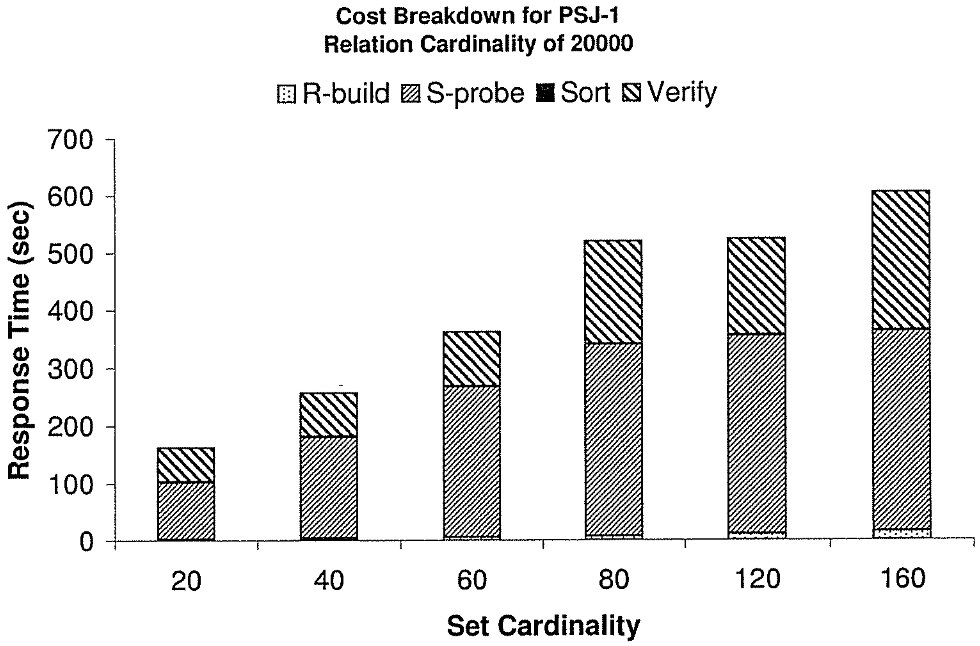


Figure 38: Cost Break Down of PSJ-1 for Relation Cardinality of 20,000

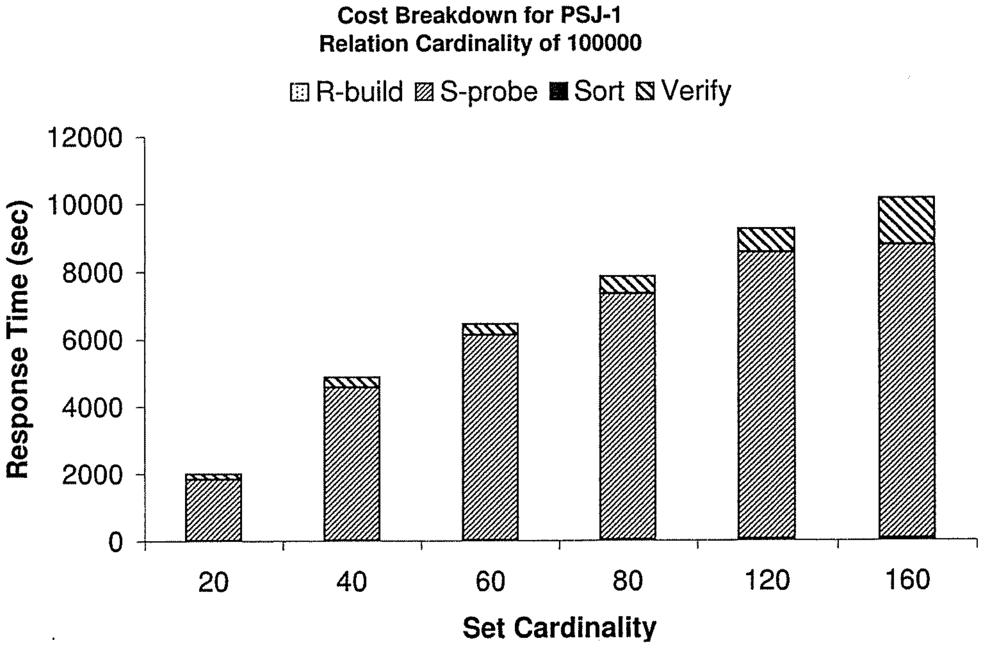


Figure 39: Cost Break Down of PSJ-1 for Relation Cardinality of 100,000

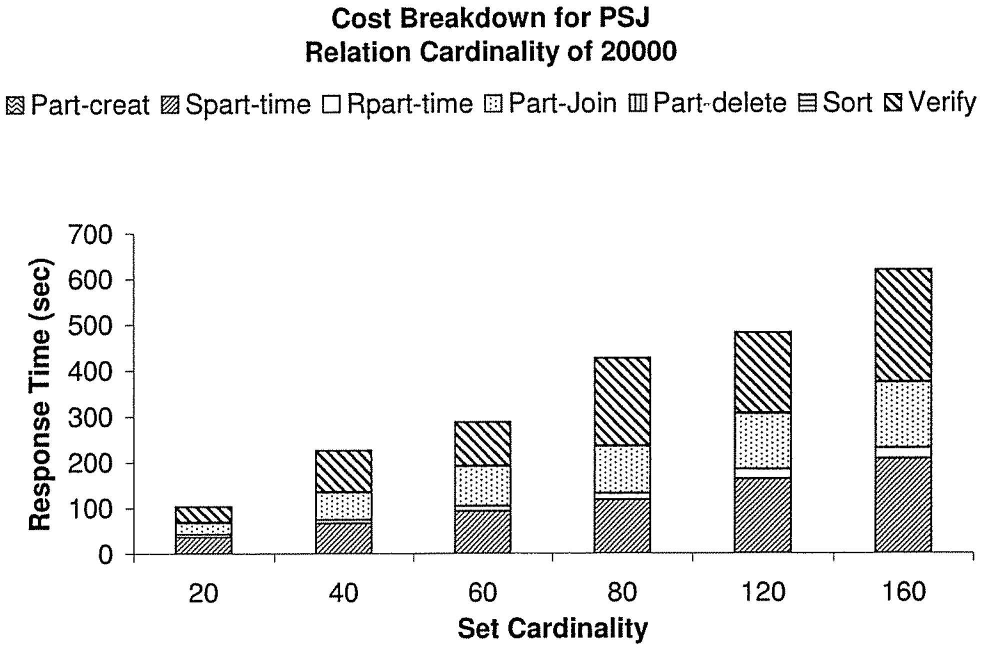


Figure 40: Cost Break Down of PSJ for Relation Cardinality of 20,000

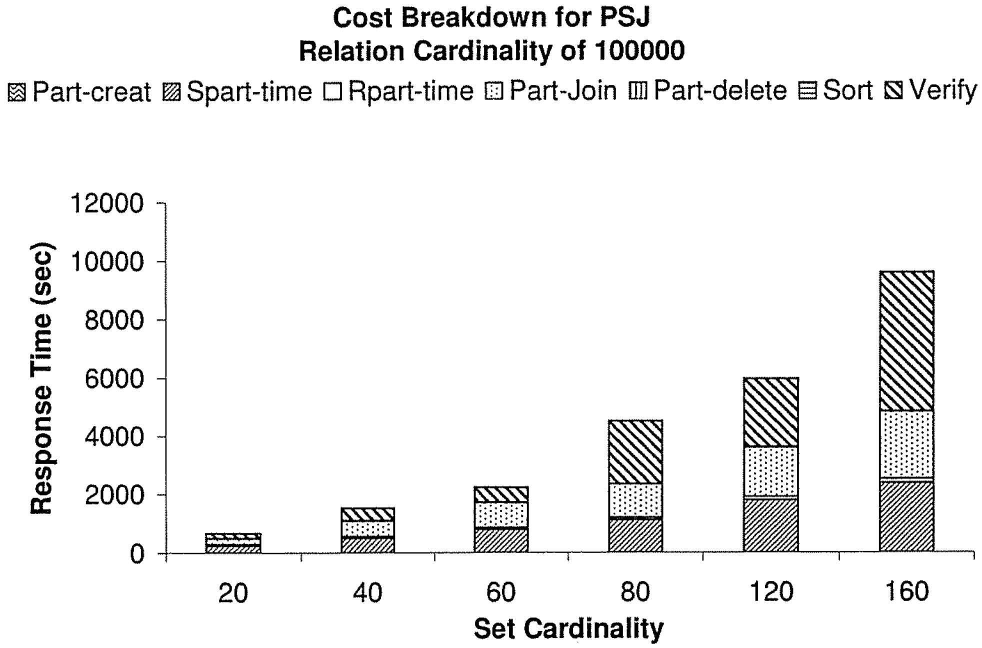


Figure 41: Cost Break Down of PSJ for Relation Cardinality of 100,000

with set cardinality as shown in Figure 40. This happens because more partitions are required and replication is higher. At the larger relation cardinality of 100,000, the set cardinality threshold beyond which PSJ-1 outperforms PSJ increases as expected as in Figure 41.

5.5 Experiment 3: Effect of Signature Size

In this experiment, we study the effect of signature size on the performance of Sig-NL and PSJ. Both algorithms use signatures for producing an intermediate candidate set of result. As noted in Section 4.3.2, the number of false drops in the candidate set is influenced by the size of the signature. Hence the choice of signature size is important both in Sig-NL and PSJ. For this experiment, we used a relation cardinality of 20,000 for both R and S, an average set cardinality of 10 for R, and average set cardinality of 20 for S. The size of domain was fixed at 10,000. For PSJ, we used the optimal number of 42 partitions, as predicted by equation (42). The result of this experiment is plotted in Figure 42.

The first observation is that for smaller signature sizes, Sig-NL is very expensive. This is because many elements in the domain hash to the same bit, thereby increasing the false drops. Such an increase in the false drops increases the time of the verification phase. As the signature size increases, the number of false drops reduces and hence the performance of Sig-NL improves. However, after a signature size of 80, increasing the signature length does not cause any significant improvement in the performance of Sig-NL. The second observation is that PSJ is relatively immune to the signature size. This is because partitioning reduces the number of false drops.

For this data set, the signature size for Sig-NL predicted by equation (30) was 173

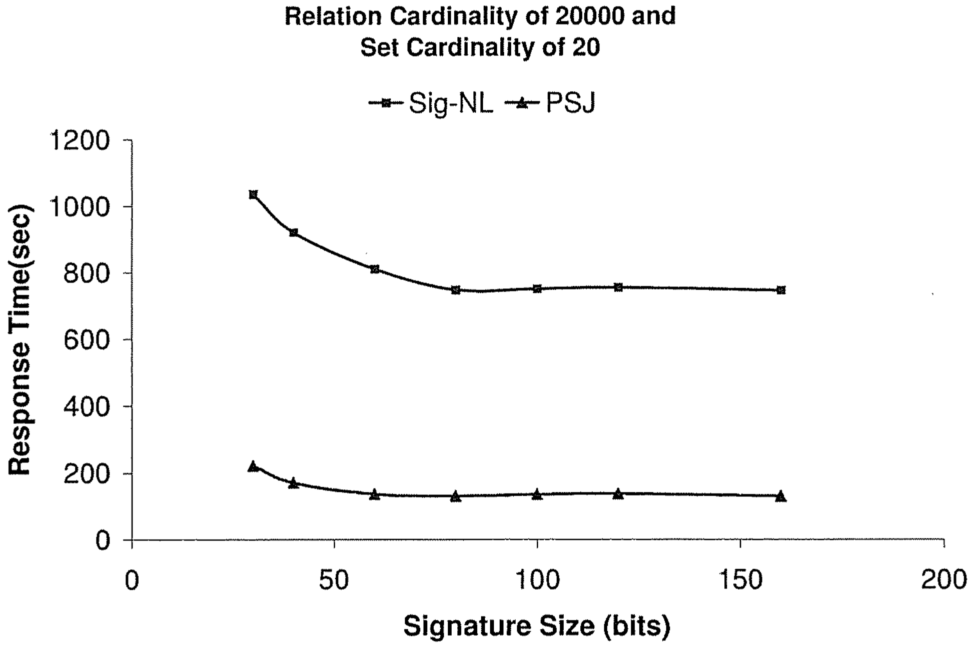


Figure 42: Effect of Signature Size

bits. For PSJ with 42 partitions, the signature size predicted by equation (48) was 116 bits. Given the flatness of the PSJ curve, it is not important to get the signature size exactly right.

5.6 Experiment 4: Effect of Increasing Partitions in PSJ

In this experiment, we study the effect of the number of partitions on the performance of PSJ. The relation cardinality of both relations was set at 20,000 and set cardinalities at 20 and 120. The set elements are drawn from a domain size of 10,000. An appropriate combination of partitions and signature size was used as determined by equations (42) and (48). The results of these experiments are shown in Figure 43 and Figure 44. Both the figures show the breakdown of total cost: partition time, partition creation and deletion

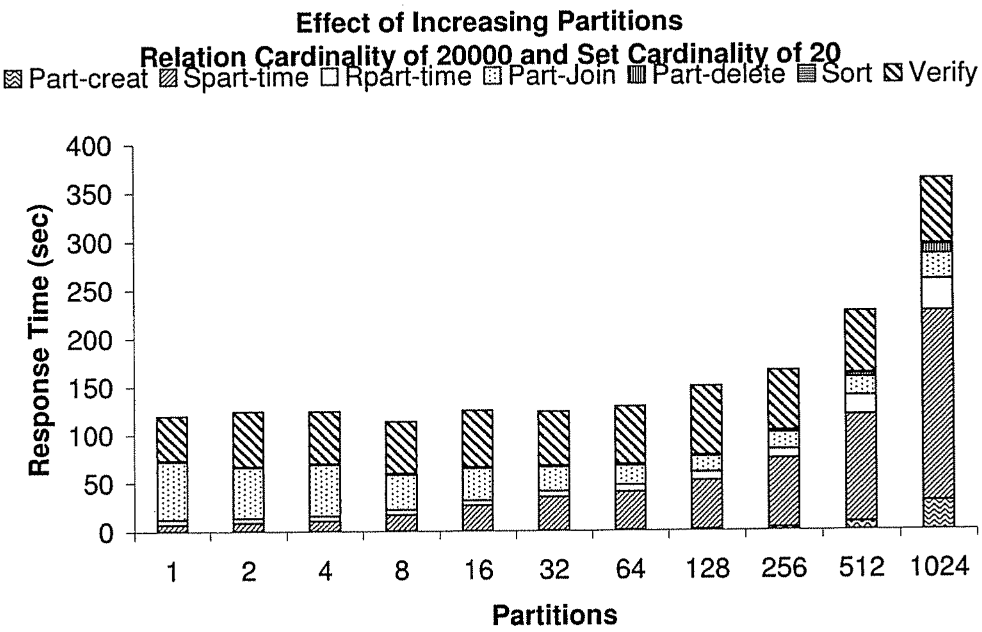


Figure 43: Effect of Increasing Partitions for Set Cardinality of 20

times, join time, sort time and verification time. From the figures, we observe that the graphs have three phases: the first phase, in which the total cost decreases gradually as the number of partitions is increased; the second phase in which the total cost is approximately constant; and the third phase in which the total cost starts increasing as the number of partitions becomes very large. In the first phase, when the number of partitions is 1, the join is essentially a signature based partition algorithm with the added overhead of partitioning. As the number of partitions increases, partitioning begins to pay off. However, this improvement in performance is not unbounded. This is when we move into the third phase. In this phase, the partitioning cost starts rising sharply, and this has an unhealthy effect on the overall join performance.

In order to further investigate the sharp increase in partitioning overhead, we plot both the total number of pages generated by the algorithm and the actual number of pages the system uses in Figure 45 and Figure 46. They show that as the number of partitions increases, there is a corresponding increase in the size of the data generated. This is

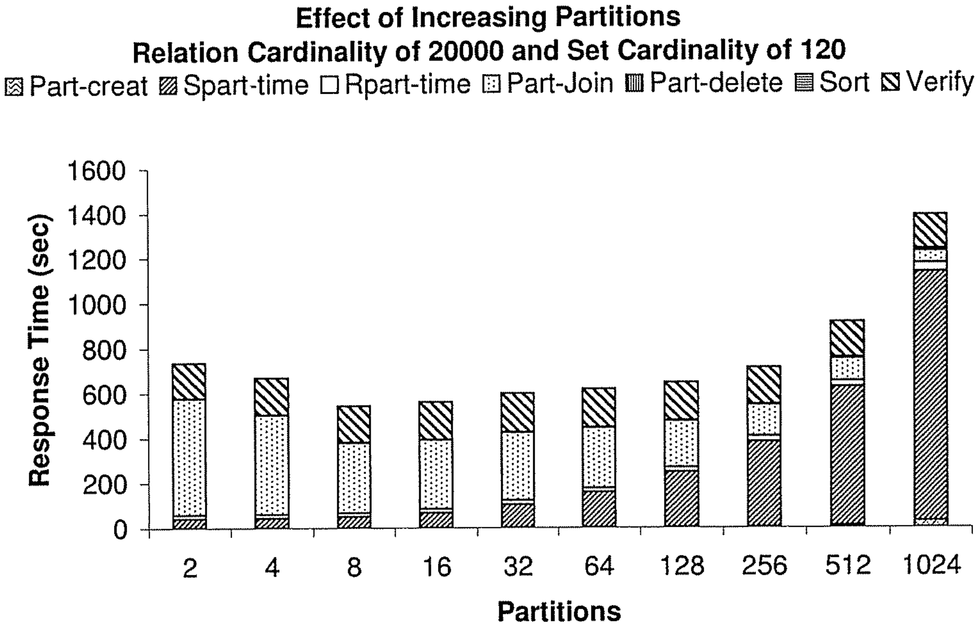


Figure 44: Effect of Increasing Partitions for Set Cardinality of 120

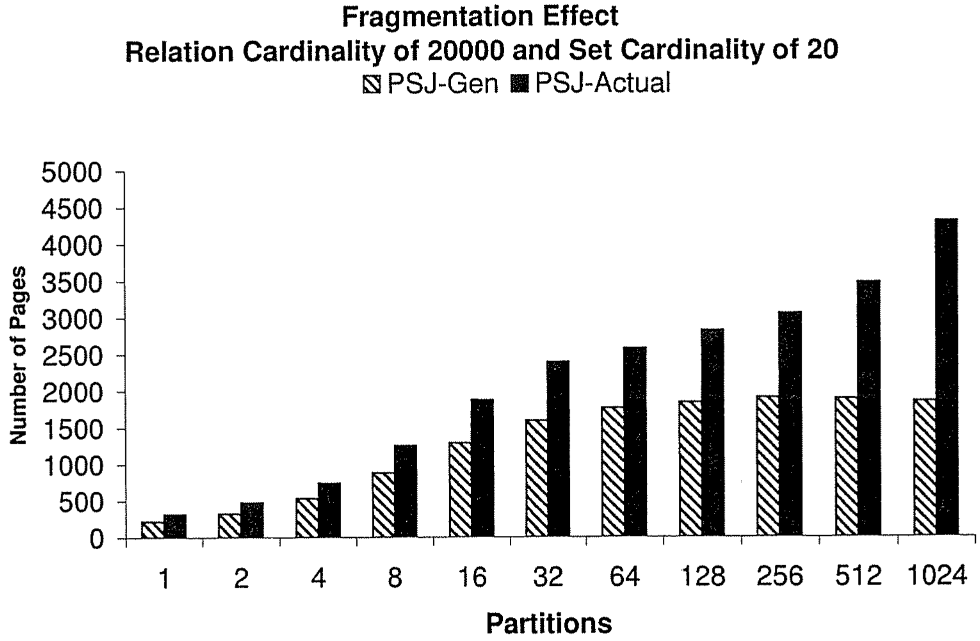


Figure 45: Fragmentation Effect for Set Cardinality of 20

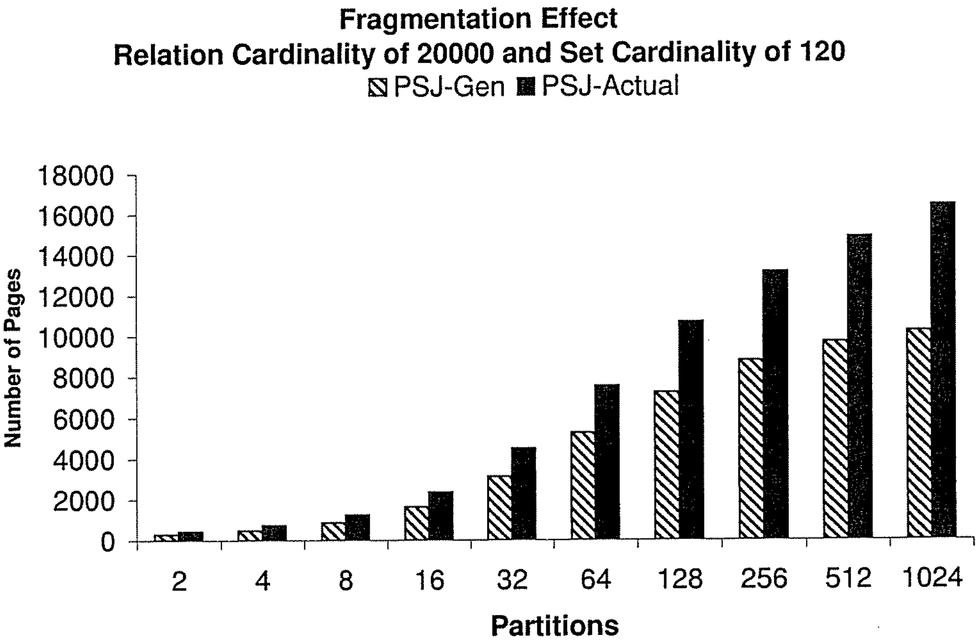


Figure 46: Fragmentation Effect for Set Cardinality of 120

because the replication of S tuples increases. However, the replication of each tuple is bounded by the set cardinality. Hence the increase in the amount of data generated flattens after 64 partitions. The actual number of pages required is substantially higher than the generated data pages. This difference is caused by per-tuple overhead and fragmentation. In addition to increasing fragmentation, the number of pins and unpins, the cost of creating and deleting the partitions also increase with the number of partitions. Thus, the partitioning overhead increases sharply when the number of partitions is large. These experiments show that the number of partitions has a critical impact on the performance of PSJ. The equation (42) can be used to estimate a reasonable number of partitions. For set cardinality of 120, the number of partitions chosen by the equation was 70.

5.7 Experiment 5: Disk Space Requirements

Here we investigate the size of the intermediate space required for Sig-NL and PSJ. We do not consider PSJ-1 since it is an in-memory algorithm. We ran experiments by varying the set cardinality for a relation size of 100,000 and also by varying the relation cardinality for a set cardinality of 120. The results are plotted in Figure 47 and Figure 48 respectively. The graph plots the number of pages generated by each algorithm and the actual number of pages used in disk. The main observation is that Sig-NL requires much less storage than PSJ as expected. The number of pages required by Sig-NL varies slightly because of the variation in signature size. Since the number of pages required by Sig-NL is so low, there is a high probability that these pages will remain in the buffer pool during the operation of the algorithm. Hence Sig-NL can be assumed to be immune from I/O cost except during the signature construction and verification phases. On the other hand, PSJ requires a large amount of intermediate storage that steadily increases as the cardinality increases. This is because a) the number of times the 3-tuple (as described in Section 4.4.3) is replicated increases as set cardinality increases and b) the number of tuples per partition increases as the relation cardinality increases.

For large data sets, the memory requirement for PSJ-1 is very high since the entire set of R signatures has to be accommodated. On the other hand, Sig-NL and PSJ adapt themselves to available amount of memory. Hence they are well suited to a multiuser environment.

5.8 Summary

This chapter investigated algorithms for computing a set containment join. These algorithms cover two possible implementations of set valued attributes: the unnested external

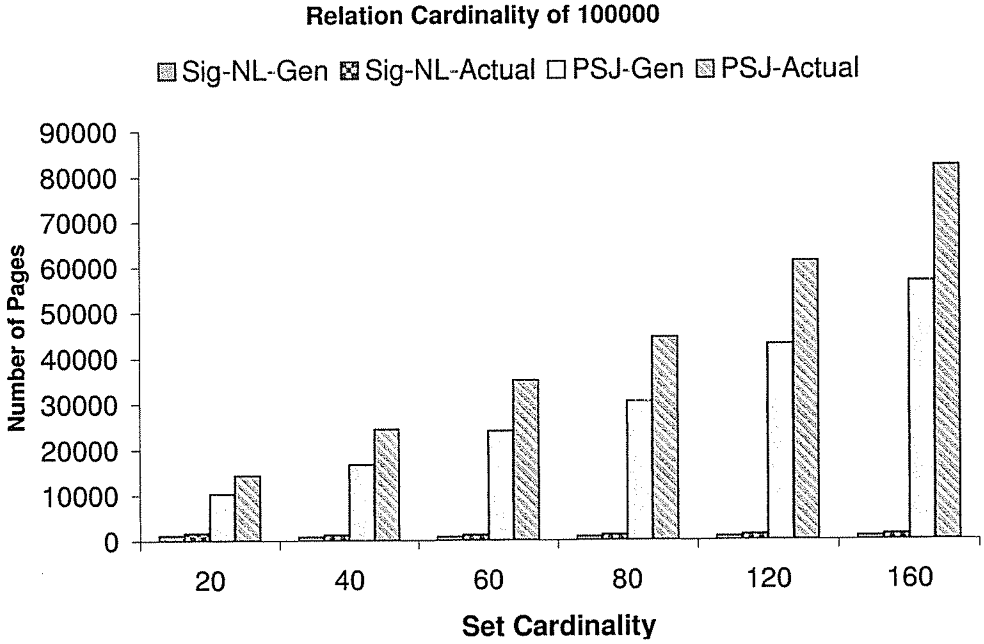


Figure 47: Intermediate Disk Space Requirements for Relation Cardinality of 100,000

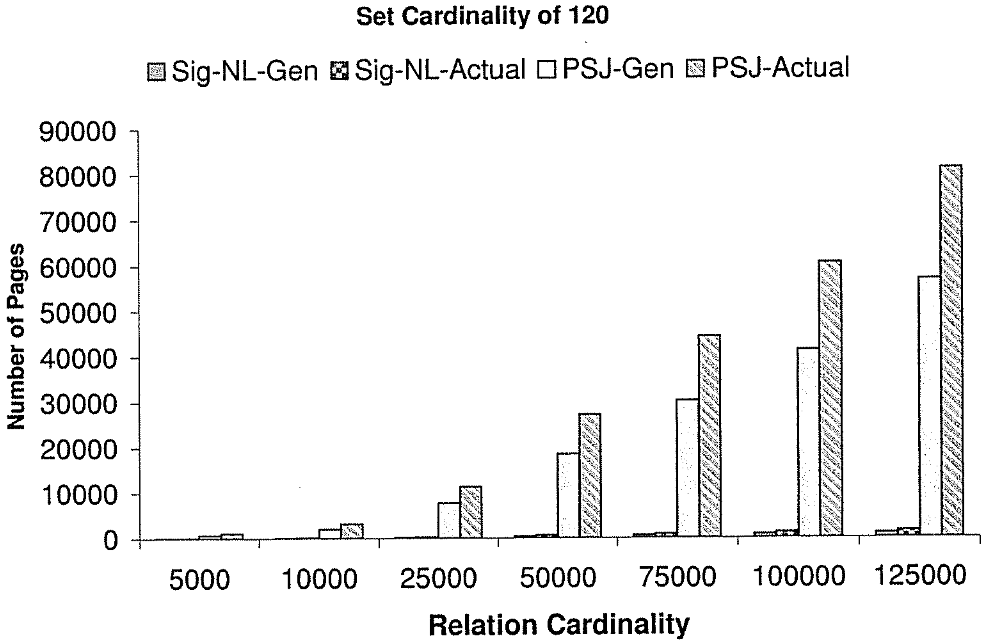
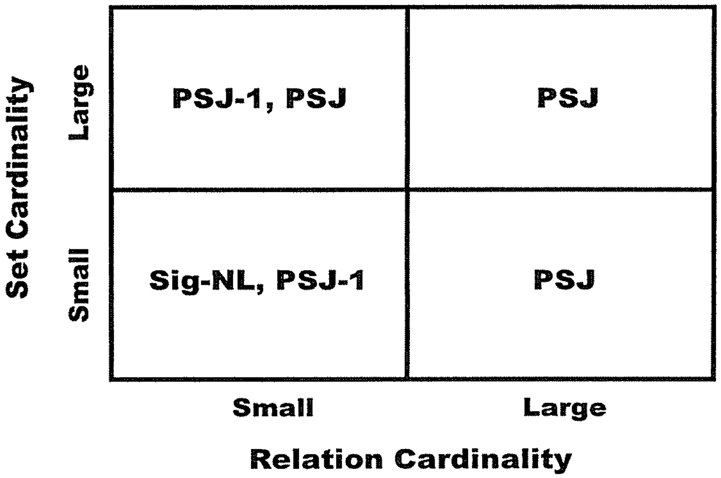


Figure 48: Intermediate Disk Space Requirements for Set Cardinality of 120



Set Cardinality	Relation Cardinality: Small	Relation Cardinality: Large
Large	PSJ-1, PSJ	PSJ
Small	Sig-NL, PSJ-1	PSJ

Figure 49: Performance Space of Set Containment Algorithms

representation and the nested internal representation. The unnested external representation is used by commercial O/R DBMSs for implementing set-valued attributes. In this case, set containment join is implemented using a standard SQL2 query. For the nested internal representation, this chapter considers two algorithms. The first is a variation of nested loops (Sig-NL) that uses signatures to speed up the evaluation of the join predicate. The second algorithm is PSJ, a new partition based algorithm that is proposed in this chapter. This algorithm is based on a two level partitioning scheme by using set elements to partition relation R and replicate relation S. Within each partition, it uses an in-memory algorithm based on partitioning of signatures.

We presented a detailed performance study of the three algorithms. The performance space of these algorithms is summarized in Figure 49. For small data sets and small set cardinalities, PSJ works well. The SQL approach and Sig-NL performs reasonably well for extremely small data sets and small set cardinalities; however, as the relation or the set cardinality size increases the performance degrades very rapidly. PSJ with one partition is usable at higher set cardinalities provided there is enough memory. Elsewhere, PSJ is the algorithm of choice.

Chapter 6

Related Work

6.1 Introduction

Sets have been studied under the context of nested relational and object oriented data models. Most of studies revolve around data modeling using sets and various kind of operations on them. There are a few implementations and performance studies that incorporate set-valued attributes. However, none of these studies consider sets under the context of object relational databases. These studies are not focused on how declarative queries using set-valued attributes can be efficiently evaluated and how their efficiency depends on the underlying organization of sets in the storage. Further most of the modeling studies outline operations on the set-valued attribute but only a few only consider how efficiently to evaluate them.

6.1.1 Chapter Organization

Related work on sets from modeling perspective are described in Section 6.2. Section 6.3 summarizes modeling and implementation under the context of nested relational databases. Section 6.4 outlines the various implementations of object oriented databases and their understanding of sets. Object relational databases are discussed in Section 6.5. Published work related with set storage organizations is described in Section 6.6. Section 6.7 discusses the reported work in signatures used in set containment algorithms.

Finally, Section 6.8 summarizes the reported algorithms for evaluating set containment joins.

6.2 Data Modeling

Sets have been studied in detail by the data modeling community. Some of the earlier semantic data models incorporate sets and collections of entities. These semantic data models have had a strong influence on nested relational and object-oriented databases. The semantic data model (SDM) [HM81] was designed for implementation as more expressive interface over a relational system. The designers attempted to distill their experience in designing databases to select the most common data semantics, and to incorporate them into a single data model. The SDM defines classes that can be related to each other through a class hierarchy. The model supports various interclass connections, which is a relationship between two classes. One of the interesting interclass connection is created by the *grouping class.* A grouping class on a class C defines a new class G with elements that are sets of elements drawn from C and establishes a relationship between C and G. This notion of grouping is applied at the type level. The collection of entities were at the top level and the relationships represented whether one is a part of the other.

DAPLEX [Shi81] is another semantic database model that uses the notation of mathematical functions as the basis upon which higher-level semantic concepts can be built. The DAPLEX considers everything as a function. Collection of entities are defined and the relationships characterize the collections. The DAPLEX query language is cast in terms of a built-in set iterators that apply a predicate to a set of values. The main operators were existential and universal quantifications.

A further discussion on the data modeling of sets can be found in [Bro81], [Bro84],

[HK87], [Gil94]. These studies discuss how sets can describe semantic notions of the real world with ease.

6.3 Nested Relational Databases

The nested relational model was introduced in [Mak77]. This work relaxes the assumption that relational attributes must be atomic and extends it to incorporate nested attributes (each attribute can be a relation). It studies the implications of such extensions to the standard relational normal forms. It introduces new definitions of functional dependencies to accommodate non 1NF forms and outlines a new definition for normal forms based on extended multivalued dependencies. A nested relational model is also proposed in [JS82]. It enriches the relational algebra with nest and unnest operations that transform between NF^2 relations and the usual ones. The algebraic properties of these operations are defined and proved. It also outlines rules which occur in combination with these operations of the usual relational algebra. Both the studies, however, never incorporated the notion of sets that require the uniqueness semantics.

Various algebraic query languages are presented in [FT83], [Zan83], [AB84], [Bid87], [DL87], [OOM87] and [Guc87]. In [Zan83] a query language called GEM was proposed in an attempt to extend the relational database model minimally. GEM adds sets as a data type, generalization, and a new data type called instance of a tuple in another relation. It extends QUEL with a range of new constructs to support operations on the data model extensions. Here the sets are viewed as a logical collection and the set operations of equality and containment are defined as part of the query language. It also notes that set operators are very expensive to support in standard relational systems. Hence it solves the problem by a two-fold approach. First, it maps subset relationships into equivalent aggregate relationships that are more efficient to support. Then it exploits the fact that set-valued attributes can be used in this capacity, so that substantial improvements in performance can be achieved by specialized storage organizations. Further, it observes that performance improvements can be obtained by declaring set-valued attributes rather than expressing using key-foreign key. Another major contribution of GEM is the clean definition of the "nested dot" notation which has been generalized and followed by several others.

[AB84] defines the VERSO model where data is organized in non First Normal Form relations. VERSO induces a hierarchical organization of data and the implicit specification of join dependencies. The nested attributes follow a recursive structure of alternating record and collection types. It also allows one to represent some simple type of incomplete information. Further, VERSO introduces four binary operations (fusion, difference, join and cartesian product) and four unary operations (projection, selection, restriction and renaming) on VERSO instances. These operations are natural extensions of relational operations and the operations of nest and unnest are very primitive sub-cases of the restricting mechanism presented. It also observes that queries which would require joins in the relational model can be realized by a filter if the joins are implicit in the VERSO format. It also shows the completeness of the VERSO algebra by showing that given a relational database schema R and its corresponding VERSO schema V, each relational query on R can be equivalently expressed by a VERSO query on V. [Bid87] further describes an algebra that takes advantage of the semantic connection among attributes implicitly specified by a VERSO schema. It shows the equivalence of VERSO and relational algebra by an extension of the tableau technique.

Extension of relational models with set-valued attributes with reference to *Statistical Databases (SDB)* have been studied in detail in [OOM87]. SDB applications usually contain aggregated data (e.g AVERAGE SALARY) qualified by set-valued attributes (e.g, JOB-GROUPS, AGE-GROUPS). Moreover, queries requesting aggregates almost always are specified over set-valued attributes. SDB extends the relational model and the associated relational algebra and relational calculus languages with set-valued attributes and aggregate functions and show that the extended languages have the same expressive power. Their study restricts relations with only simple- or set-valued attributes (i.e., set-valued relations), which are subset of N1NF relations. Their main contribution is extending [Klu82] (which discusses translating calculus expressions into algebra expressions) with set-valued attributes, which is to obtain algebraic expressions for calculus objects and then recursively combine these expressions until a final algebraic expression equivalent to the original calculus expression is obtained. They also define a new set of operators called "pack", "unpack" and "aggregation-by-template." Pack is similar to nest, except that it performs a set union operation, while nest adds another level of nesting to the attributes.

Query optimization for the nested relational algebra is discussed in [Sch86], [DG87], [Col89], and [KP90]. Design and normalization issues are discussed in [OY87] and [RK87].

There are various implementations of nested relational database systems. [DKA$^+$86] and [PD89] describe the implementation of Advanced Information Management Prototype at the IBM Heidelberg Scientific Center. This implementation supported type constructors (lists, sets and tuple), support for user-defined types, and functions. It also supported the standard set of nested algebra operations. The database server consisted of various components such as: *buffer and segment manager* — for managing the memory and disk, *sub-tuple manager* — allows access to data in terms of records, *complex object manager* — for assembling complex objects from tuples and the query processor. The complex object manager is aware of the physical structure of the complex objects and provides a uniform view of them independent of physical representation. The nested structures are stored in separate table for each instance and tuple identifiers are used as pointers to reflect the nested nature.

The DASDB project at the Technical University of Darmstadt [SS89] supported the nested relational algebra with nest and unnest operations. The complex objects or nesting structures are stored in hierarchical cluster fashion. A complex object is stored in as few pages as possible and stored sequentially so that disk seeks are minimized. Motivation for such a storage scheme arises from the fact that one can do better than just storing tuples of a relation one by one and external storage devices provide a block structured linear address space. Hierarchies are the most general structures that can be linearized (without introducing redundancy or using pointers). DASDB supports algebraic and physical optimization and an SQL-like query language.

The VERSO database machine [SAB$^+$89] was attempted to justify the approach consisting in relegating tasks to a processor close to the mass storage device and checking whether an automaton-like mechanism for this on-the-fly filtering capability is useful. The major motivation was to speedup the performance of a relational DBMS. However, as described earlier, it implemented a nested relational model. The VERSO system contained three layers: the top level recognized the VERSO relations, the second level was the physical representations of the VERSO relations in terms of files and indices and the final level was the disk block characterized by its address. The index is a non-dense index which stores for each block, the smallest tuple of the block (not its key) followed by the block address. The stored information is compacted through a trie structure. The query processor uses this index to efficiently access the data.

The ANDA project at Indiana University is described in [DG88] and [DG89]. They describe a single tree index which is used as a storage structure for the entire database.

Typically, indices are built on all or some of the attributes that maps the values to list of tuple identifiers. ANDA uses a domain based approach (VALTREE) in which an atomic value maps to a list of tuple identifiers of tuples in all relations in the database that contain that value. This is generalized to include the list of identifiers in all structures and sub-structures. The VALTREE is made up of five different levels: domain, value, attribute, structure and identifier levels. A cache was used to speedup the operations in this tree. The advantage of the VALTREE is that given a value, it provides fast access to tuple identifiers containing all occurrences of the value throughout the entire database. It also supports the regular data manipulation and maintenence operations.

SQL-like languages for nested relational queries are presented in [Bra83], [PA86], [PT86], [RKB87] and [RK87]. The use of the relational and nested relational models to describe text databases, user interfaces, and operating system is discussed in [HSW83], [SWH83], [BHH+84], [KKS88], and [Kor86]. A collection of recent papers on nested relations appears in [AFS89].

6.4 Object Oriented Database Systems

As object oriented languages were getting popular, there were lot of attempts to add persistence to these languages. The argument for this approach was that some applications just need to manage permanent data, and would be happy with the imperative programming model of such a language only if its type system were available for use in constructing complex persistent structures. Since programming languages support rich collection constructs, there was a need to support full-fledged collections. But they were restricted to just storage and retrieval of collections with the exception of Objectstore that supported declarative querying. A good survey of such related work can be found in [AB87].

Another approach to satisfying the need for non-traditional database applications combines all the features of a modern database system with those of an object oriented language, yielding an object-oriented database (OODB) system. Such a system supported many collection types including sets. Three early OODB projects laid the foundation in this area - Gemstone [CM84] [MS87] which was based on Smalltalk, Vbase [AH87], which was based on a CLU-like language, and Orion [BCG$^+$87], which was based on Common LISP Object System (CLOS). Such systems supported various features like complex objects, object identity, encapsulation, inheritance and substitutability, late binding, computationally complete methods, an extensible type system, persistence, secondary storage management, concurrency control, recovery, and ad hoc queries. New SQL-like languages were designed to support powerful querying. These querying constructs use collections as first class type. These query languages allowed nested queries, universal, and existential quantification queries.

The Iris Object-oriented database management system [FBC$^+$90] provided support for object, object properties, object operations and rules, types and type hierarchies. Type extents provided the collection semantics but typical collection operations were not supported. Iris supported an Object SQL interface that provided direct references to objects and the ability to invoke user defined functions anywhere in the SELECT and WHERE clauses. It used a relational storage engine and provided a translation interface mechanism for converting the records into complex objects.

O_2 [LRV88] is an another object oriented database system. The O_2 data model consisted of objects that have identity and can encapsulate data and behavior. They supported types and classes and the types are recursively constructed using atomic types and type constructors (sets, list and tuple constructors). O_2 makes a distinction between values and objects. Objects have identity while values do not. A value can be embedded in another value or in an object. A type describes a minimal behavior for an object. Subtypes (and their relationships) are defined explicity by the user. The query language supported two modes: one accessible from a programming language in which encapsulation is enforced; and the other for ad hoc querying using declarative language (an extension of SQL) in which encapsulation is relaxed. It is functional and first order. SQL syntax was used to filter lists and sets with an option for specifying predicates using universal and existential quantifiers. The object manager used WiSS [CDKK85]. The tuples were stored as records in a page. Lists were implemented as ordered trees. Sets were stored as objects which have a set of pointers. Indices are allowed to be created in sets to speedup the membership tests. O_2 objects cannot be deleted explicity; instead references to them are deleted and the objects are garbage collected.

ORION was a series of object-oriented database systems that were prototyped at MCC. The implementation is described in detail in [BCG$^+$87], [BKKK87], [BKK88], [Kim89], [Kim90] and [KGBW90]. The first version of ORION provided persistence to CLOS. The second version of ORION provided support for objects, object identity, classes and collections. However, collection operators were never defined. The relationship among objects were explicitly qualified by the nature of dependency (shared vs. exclusive). The query language was SQL-like but used a CLOS syntax. The major distinction between other systems and ORION is that it supported an excellent framework for dynamic schema evolution. Objects are stored in a storage format that contained the unique identifier of its class and its own unique identifier. Collections were stored as a set of references within each object.

Gemstone [MSOP86], developed at Servio, belongs to the class of systems that provided persistence to object oriented languages. It added persistence to Smalltalk. Gemstone incorporates object identity and encapsulation via data abstraction which defines an external interface as a set of messages. It supports single inheritance and collection types. Also, it is a disk based and provides regular database features (concurrency control, recovery, index, and querying). Gemstone supports a query language but queries are formed over the instance variables of an object. Gemstone supported various indexing strategies over collections: single step and multi-step. Collections were implemented using references to allow sharing. Objects in collections can be one of the descendants of the type, the collection is made of.

Further overviews of object oriented databases can be found in [Kim95] and [ZM90].

6.5 Object Relational Database Systems

Object-relational database systems belong to the class of extended relational systems, which try to subsume both relational and object features. The manifesto of [Aea89] provided the main tenets: provide support for rich object structures and rules, subsume second generation (i.e., relational) DBMS, and to be open to other subsystems, e.g., tools and multi-database middleware products. The manifesto further elaborates the features such a system should support: a rich type system, inheritance, functions and encapsulation, optional unique ids, and rules/triggers; a high level query-based interface, stored and virtual collections, updatable views, and separation of data model and performance features; accessibility from multiple languages, layered persistence-oriented language bindings, SQL support, and a query-shipping client/server interface. These systems typically start from a relational model and its SQL language and build from there. Early systems supported two types of objects ADTS, row types and collection types. Thus, the top level of an object-relational database schema is still a collection of named relations. However, the objects in the relations can now be as rich as those supported by OODB systems.

In order to support the optimization of object relation queries, traditional relational optimizers have to be extended to accommodate the new set of features required. [Sto96] provides an enumeration of the desired set of extensions. The majority of them allow the user to specify properties about the newly added types: user-defined selectivity functions, user-defined comparison operators, and user-defined commutators and algebraic rewriting for sets. One of the important issue is the optimization of expensive functions. Predicate migration [HS93] is a cost based technique that allows the placement of expensive functions in the join trees such that the query is least expensive to compute. Runtime optimization techniques such as re-optimizing the query in the middle of execution is proposed in [KD98]. The motivation for this kind of optimization is that statistics for user defined methods are impossible to maintain, and if maintained could get outdated leading to potentially expensive plans. Statistics collectors are added in the query plans at various points. A re-optimization takes place when the size of the intermediate results is far apart from the estimated set of results at that stage. This technique might be useful if there is a cascade of set containment joins since the statistics for containment joins is a hard problem.

The best known research implementations of O/R DBMS are POSTGRES [SK91] [Sto87] from University of Berkeley and Paradise [PYK$^+$97] from University of Wisconsin. POSTGRES was further commercialized as Illustra which was bought by Informix. It supported the dynamic addition of new types, support for complex objects including set-valued attributes, inheritance and rules support and provides an extension of QUEL as the language. Further, POSTGRES allows for addition of new index types for new user defined data types and allows for addition of various comparison operators. It also provides a fast path to directly interface the query in parse-tree to the optimizer. The storage manager uses the idea of "no-overwrite" rather than the typical write-ahead-log (WAL). Using this technique, the old record remains in the database when an update occurs. Consequently, POSTGRES has no log and it is simply two bits indicating whether each transaction committed or aborted or in progress. The nice features of such a system is instantaneous crash recovery and time travel. In such systems, the sets were still stored as standard relation types.

Paradise departs from POSTGRES in that it is a parallel object relational database system. The main contribution of Paradise is to explore the parallelization of object relational features in a shared nothing environment. As required, it supports a subset of SQL with object relational extensions. It provides facilities for compile time addition of user defined types with support for arrays, image, audio and video data types. In addition, Paradise supported a full set of spatial data types and built-in spatial operations to speedup spatial queries. The impact of large objects on parallelization lead to the hybrid model of "push" for tuples and "pull" for assembling large objects. The storage manager used by Paradise is SHORE [CDN+94] which supports object storage, transactions and recovery and a full set of indices ranging from B-Tree, R-Tree and bitmap indices.

Commercial implementations of such systems are available from IBM (IBM-UDB), Oracle 8 & 9 and Informix. A nice survey of the evolution of object oriented and object relational systems is presented in [CD97].

6.6 Set Storage Representations

Relational systems store the data in a Normalized Storage Model (NSM) where all the attributes constitute the structural representation of the tuple. Early research on decomposing a tuple into multiple attributes and storing them independently for relational systems is described by Copland [CK85]. The Decomposition Storage Model (DSM) vertically partitions a relation and stores each attribute in a file of its own. To facilitate the reconstruction of a tuple, a surrogate value is assigned and is replicated for every instance of attribute of the tuple. The DSM has been compared with NSM using various criteria: relative complexity, storage requirements, and update and retrieval performance. However, such an approach is impractical since it provides dismal update performance. Such a feature is not desirable in a relational system where updates are frequent.

Valduriez [VKC86] describes a hybrid storage scheme called partial DSM (P-DSM) that vertically partitions a relation based on access patterns and the frequency of accesses of the attributes. Thus the storage structures are organized by exploiting the query workload presented to the system. The P-DSM scheme leads to replication when attributes are stored in multiple partitions, leading to update penalties. The focus of their work is to characterize the performance issues of these storage schemes under the context of programming environments that manipulate complex objects. In such environments, the predominant type of access is to assemble the sub-objects of a given complex object. Related work on the quantitative evaluation of the number of disk I/Os for a complex object application can be found in [TRSB93]. The measurements are reported from the DASDBS storage system for complex objects [SPSW90]. The set of queries considered for the benchmark includes the retrieval of a complex object, and insertion and update of sub-objects. It concludes that a variation of DSM that clusters sub-objects belonging to the same object performs the best.

6.7 Signatures

Signatures have been studied in detail in [FC84] and [ZMR98] under the context of fast retrieval of documents matching a query predicate. [IKO93] studies the application of signatures for evaluating conjunctive and disjunctive predicates over set-valued attribute.

6.8 Set Containment Joins

Since join is one of the most important operators in relational database system, it has received major attention in the literature. In the relational domain, various algorithms for join have been proposed. Initially sort merge join [BK76] was proposed as an improvement over naive nested loops. Later, partition based hash joins [Bra84] [DKO$^+$84] were proposed to improve upon a sort merge join. All the above algorithms tackle the problem when the join predicate is an equality between atomic values. Non-equality joins have been considered in detail in [DNS91]. Pointer joins for efficiently traversing path expressions in object oriented databases have been studied in [DLM93] and [SC90]. Various joins have been proposed in the spatial domain for computing join predicates that check whether the objects overlap or are contained one within the other. Partition based spatial joins are investigated in [LR96], [PD96].

There is very little attention given to set containment joins in the literature. The only reported work on set containment joins the author is aware of are: [HM96] and [HM97]. These papers investigate in detail naive nested loops and signature nested loops and proposes a new algorithm called signature hash join. Signature hash join operates by splitting the signatures of relation R into a set of partitions based on a subset of bits in the signature. Using a subset capable hash function, for every tuple in S, it generates all possible subset signatures (exponential in nature) and probes the signature hash table to find the joining pairs. Furthermore, it proposes how to identify this subset of bits such that the hash table will be able to fit in memory and the false drops are kept to minimum. Even though it provides excellent speed up, the main drawback of this algorithm is that it requires the entire signature hash table to fit in memory. Hence it does not scale very well to larger data sets. However, it can serve as an excellent candidate for joining individual partitions in a partition based set containment algorithm where the partitions are small enough that they can be held in memory.

Chapter 7

Conclusion

This dissertation explored a range of issues that rise in incorporating set-valued attributes in object relational databases. The major issues studied in detail are the efficient options for storing set valued attributes and faster evaluation of set containment joins. We showed that the nested form of storage representations are compelling by demonstrating the faster performance of conjunctive and disjunctive queries over these representations. We further showed that with the addition of new operators set containment joins can be efficiently evaluated on the nested representations.

7.1 Contributions

The contributions of this dissertation are (a) a detailed performance evaluation of storage representations for sets, and (b) an efficient algorithm for set containment joins. In Chapter 2, we analyzed the space of representations by classifying them based on two orthogonal directions: *nesting* and *location.* Based on this classification, we evaluated the storage representations nested internal, unnested external and nested external under the context of conjunctive and disjunctive queries. We also evaluated their indexed variants by augmenting each of the representations by either a simple index for unnested representations or an unnested index for nested representations. We developed a detailed analytical model in order to compare the cost of evaluation of conjunctive and disjunctive queries.

In Chapter 3, we verified the analytical model with an implementation in Paradise. While common "folk wisdom" suggests that small sets should be stored internal with their containing tuples and large sets should be stored externally, experiments with our implementation show that the decision is not that simple. All the experiments conclude that nested representations are better than unnested representations. This is because the cost of bringing the tuple into memory and processing it is amortized over all set elements for nested representations. Also, evaluation of set predicates can prematurely terminate without looking all the elements. Indexed representations provide faster lookup; however, the usability of indices is limited to a particular range of selectivities. The usable selectivity range of indexed unnested representation is much lower than the nested ones. Even though the nested representations provide faster performance, the major drawback is that they require additions and modifications to the storage and query evaluation engine.

In Chapter 4, we proposed various algorithms for evaluating the operation of set containment – the SQL query approach and signature nested loops. Further, we developed a new algorithm called Partition Set Join Algorithm (PSJ) for efficiently computing set containment joins. The partition set join algorithm (PSJ) operates on nested representations while the SQL query approach operates on unnested external representations. Further, the settings for the required parameters of the number of partitions and signature size were examined in detail.

The performance of these algorithms are evaluated by an implementation in Paradise. Our performance study shows that for small relation cardinalities and small set cardinalities, PSJ works well. The SQL approach and Sig-NL perform reasonably well in comparison with PSJ for extremely small data sets and small set cardinalities; however, as the relation or the set cardinality increases, the performance degrades very rapidly. A variation of PSJ with one partition performs better at higher set cardinalities if sufficient main memory is available to hold the signatures of relation R. In the rest of performance space, PSJ outperforms the other algorithms.

7.2 Future Work

There exists a lot of room for further research in the area of set valued attributes. This work can be broadly categorized as follows:

7.2.1 Heterogenous Representations

In Chapter 2, we restricted our attention to the homogenous representations, in which the set instances of all tuples in a relation use the same type of representation. However, such restriction can be relaxed, and each set instance can choose their own choice of the representation. The choice can be based on various characteristics: set cardinality, the nature of operations performed, and access frequency of the set instance. It will be important to analyze whether such representations are required for practical data sets. Also, it will be interesting to explore the issues that arise in storing set instances in heterogenous form and study in detail the related issues in query evaluation and optimization.

7.2.2 Set Intersection Joins

The PSJ algorithm for set containment can be easily extended to evaluate set intersection. The major extensions required are that, instead of replicating a single relation both the relations need to be replicated. The number of partitions and the chosen signature sizes might play a more critical role here than in set containment joins.

7.2.3 Declustering of Set Valued Attributes

Our work explored the various possibilities of storing sets in a single server environment. However, the tradeoffs could be different when sets are implemented in a parallel shared nothing environment. The problem of declustering tuples with set-valued attributes needs to be addressed. There are various approaches for declustering with various tradeoffs and pros and cons. Some of the declustering strategies include

- **Signature Declustering** — Compute the signature of the set and, based on the signature value, a node is chosen. For conjunctive and disjunctive queries, a query signature is formed using the predicate. This query signature is used to compute the potential set of nodes that requires to be searched. The disadvantages of this approach are (a) many nodes have to be searched even for singleton queries (b) in a multiuser environment the set of nodes used in search might overlap, thereby increasing execution skew in the overlapping nodes.

- **Unnested Declustering** — This is a straight forward approach in which the sets are stored in unnested form and one of the traditional declustering methods of round robin, hash and range partitioning can be used. This simple approach might be attractive since all the queries potentially translate into joins and have to be executed across all the nodes. Hence the load would be balanced across all the systems. However, the potential disadvantage is that response time might be affected because of the huge joins even for simple disjunctive and conjunctive queries.

As a consequence of parallel declustering, it will be interesting to see how the set containment and intersection join algorithm adapt in these parallel environments.

7.2.4 Result Size Estimation for Queries with Set Predicates

The third area of research involves estimating the result size of a query when the predicate involves set valued attributes. Histograms and sampling have been traditionally used for estimating result size. The main idea in histograms is to capture the occurrence frequency of attribute values and decide which attribute values had to stored in the statistics. With set-valued attributes, things get much more complicated.

7.3 Summary

This dissertation considers the implementation of set-valued attributes in object relational database systems. We have focused on the options for representing sets, and on a particularly challenging operation over sets, the set containment join. We have shown that the implementation of the nested representations for set storage, along with our new algorithm for set containment joins, can provide substantially better performance than that given by the currently popular "translate to standard relations" approach.

Bibliography

[AB84] S. Abiteboul and N. Bidoit. Non-first normal form relations to represent hierarchically organized data. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 191–200, 1984.

[AB87] M. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2), 1987.

[Aea89] M. Atkinson and et al. The object-oriented database system manifesto. In *Proceedings of the 1st DOOD Conference*, Kyoto, Japan, 1989.

[AFS89] S. Abiteboul, P. C. Fischer, and H. J. Schek, editors. *Nested Relations and Complex Objects in Databases.* Lecture Notes in Computer Science 361. Springer-Verlag, 1989.

[AH87] T. Andrews and C. Harris. Combining language and database advances in an object oriented development environment. In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, Orlando, FL, October 1987.

[BCG+87] J. Banerjee, H. T. Chou, J. Garza, W. Kim, D. Woelk, and N. Ballou. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1), 1987.

[BHH+84] A. Bernstein, J. Heller, P. B. Henderson, Z. M. Kedem, E. Sciore, D. S. Warren, L. D. Wittie, and A. Zorat. A data oriented network system. Technical Report 84/091, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY, 1984.

[Bid87] N. Bidoit. The VERSO algebra or how to answer queries with fewer joins. *Journal of Computer and System Sciences*, 35(3):321–364, December 1987.

[BK76] M. W. Blasgen and K.P.Eswaran. On the evaluation of queries in a relational database system. Technical report, IBM, 1976.

[BKK88] J. Banerjee, W. Kim, and K. C. Kim. Queries in object oriented databases. In *Proceedings of the International Conference on Data Engineering*, February 1988.

[BKKK87] J. Banerjee, W. Kim, H. J. Kim, and H. F. Korth. Semantics and implementation of schema evaluation in object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 311–322, 1987.

[Bra83] J. Bradley. Application of SQL/N to the attribute-relation associations implicit in functional dependencies. *International Journal of Computer and Information Sciences*, 12(2):65–86, 1983.

[Bra84] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 323–333, 1984.

[Bro81] M. L. Brodie. Association: Database abstraction. In *Information Modeling and Analysis*, pages 583–608. North Holland, 1981.

[Bro84] M. L. Brodie. On the development of data models. In *Conceptual Modeling*, pages 19–47. Springer Verlag, 1984.

[Cat94] R. G. G. Cattell. *The Object Database Standard: ODMG 93*. Morgan Kaufmann, 1994.

[CD97] M. J. Carey and D. J. DeWitt. Of objects and databases: A decade of turmoil. In *Proceedings of the International Conference on Very Large Databases*, pages 1–12, Mumbai, India, 1997. Invited Paper.

[CDKK85] H. T. Chou, D. DeWitt, R. Katz, and A. Klug. Design and implementation of the Wisconsin Storage System. *Software Practice and Experience*, 15(10), 1985.

[CDN$^+$94] M. Carey, D. DeWitt, J. F. Naughton, M. Solomon, and et al. Shoring up persistent applications. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Minneapolis, MN, June 1994.

[CDN$^+$97] M. Carey, D. Dewitt, J. F. Naughton, M. Asgarian, P. Brown, J. E. Gerke, and D. N. Shah. The BUCKY object-relational benchmark. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1997.

[CK85] G. P. Copeland and S. Khoshafian. A decomposition storage model. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 268–279, Austin, Texas, May 1985.

[CM84] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1984.

[Col89] L. Colby. A recursive algebra and query optimization for nested relations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 273–283, 1989.

[DG87] A. Deshpande and D. Van Gucht. A storage structure for unnormalized relations. In *Proceedings of the GI Conference on Database Systems for Office Automation, Engineering and Scientific Applications*, pages 481–486, Darmstadt, 1987.

[DG88] A. Deshpande and D. Van Gucht. An implementation for nested relational databases. In *Proceedings of the International Conference on Very Large Databases*, pages 76–87, Los Angeles, USA, 1988.

[DG89] A. Deshpande and D. Van Gucht. A storage structure for nested relational databases. In S. Abiteboul, P. C. Fischer, and H. J. Schek, editors, *Lecture Notes in Computer Science*, pages 69–84. Springer-Verlag, 1989.

[DKA$^+$86] P. Dadam, R. Kuespert, F. Anderson, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walsh. A DBMS prototype to support extended NF2 relations: An integrated vied on flat tables and hierarchies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 356–366, 1986.

[DKO$^+$84] D. Dewitt, R. Katz, F. Ohlken, L.Shapiro, M.Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 1–8, 1984.

[DL87] V. Deshpande and P. Larson. An algebra for nested relational databases. Technical Report CS-87-65, Department of Computer Science, University of Waterloo, Ontario, Canada, 1987.

[DLM93] D. Dewitt, D. Lieuwen, and M. Mehta. Pointer-based join techniques for object-oriented databases. In *PDIS*, 1993.

[DNS91] D. Dewitt, J. Naughton, and D. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *PDIS*, Miami Beach, 1991.

[FBC$^+$90] D. Fishman, D. Beech, H. Cate, E. Chow, T. Connors, J. Davis, N. Derrett, C. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. Neimat, T. Ryan, and M. Shan. IRIS: An object oriented database management systems. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[FC84] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Office Information Systems*, 2(4):267–288, October 1984.

[Fel57] W. Feller. *An Introduction to Probability Theory and Applications*, volume 1. John Wiley and Sons, 1957.

[FT83] P. C. Fischer and S. Thomas. Operators for non-first-normal-form relations. In *Proceedings of the International Computer Software Applications Conference*, pages 464–475, 1983.

[Gil94] M. M. Gilula. *The Set Model for Database and Information Systems.* Addison-Wesley Publishing Company, 1994.

[Guc87] D. Van Gucht. On the expressive power of the extended relational algebra for the unnormalized relational model. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 302–312, 1987.

[HK87] R. Hull and R. King. Semantic database modeling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[HM81] M. Hammer and D. McLeod. Database description with SDM: A semantic data model. *ACM Transactions on Database Systems*, 6(3), 1981.

[HM96] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. Technical report, University of Mannheim, 1996.

[HM97] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Athens, Greece, 1997.

[HO88] A. Hafez and G. Ozsoyoglu. The partial normalized model of nested relations. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 100–111, Los Angeles, California, September 1988.

[HS93] J. Hellerstein and M. Stonebraker. Predicate pushdown for expensive functions. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Philadelphia, PA, May 1993.

[HSW83] P. B. Henderson, E. Sciore, and D. S. Warren. A relational model for operating system environments. In *Proceedings of the DEC Workshop*, 1983.

[IKO93] Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evaluation of signature files as set access facilities in OODBS. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 247–256, Washington, D.C., 1993.

[JS82] G. Jaeschke and H. J. Schek. Remarks on the algebra of non first normal form relations. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 124–138, 1982.

[KD98] N. Kabra and D. DeWitt. Efficient mid-query reoptimization of sub-optimal query execution plans. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Seattle, June 1998.

[KGBW90] 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):109–125, March 1990.

[Kim89] W. Kim. A model of queries for object oriented databases. In *Proceedings of the International Conference on Very Large Databases*, 1989.

[Kim90] W. Kim. Architectural issues in object-oriented databases. *Journal of Object Oriented Programming*, March/April 1990.

[Kim95] W. Kim. *Modern Database Systems: The Object Model, Interoperability and Beyond.* ACM Press, 1995.

[KKS88] H. J. Kim, H. F. Korth, and A. Silberschatz. PICASSO: A graphical query language. *Software-Practice and Experience*, 18(3):169–203, March 1988.

[Klu82] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of ACM*, 29(3):699–717, July 1982.

[Kor86] H. F. Korth. Extending the scope of relational languages. *IEEE Software*, 3(1):19–28, January 1986.

[KP90] H. F. Korth and X. Peltier. Query processing issues in knowledge bases. In *Proceedings of the Conference on Artificial Intelligence in Petroleum Exploration and Production*, 1990.

[LR96] M. Lo and C. Ravishankar. Spatial hash-joins. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Montreal,Quebec, May 1996.

[LRV88] C. Lecluse, P. Richard, and F. Velez. O_2, an object-oriented data model. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 424–433, Chicago, May 1988.

[Mak77] A. Makinouchi. A consideration of normal form on not necessarily normalized relations in the relational model. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 447–453, 1977.

[MS87] D. Maier and J. Stein. Development and implementation of an object-oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 355–392. MIT Press, Cambridge, MA, 1987.

[MSOP86] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented DBMS. In *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 472–482, 1986.

[OOM87] G. Ozsoyoglu, Z. M. Ozsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems*, 12(4):566–592, December 1987.

[OY87] G. Ozsoyoglu and L. Yuan. Reduced MVDs and minimal covers. *ACM Transactions on Database Systems*, 12(3):337–394, September 1987.

[PA86] P. Pistor and F. Andersen. Designing a generalized NF2 model with an SQL-type language interface. In *Proceedings of the International Conference on Very Large Databases*, pages 278–285, 1986.

[PD89] P. Pistor and P. Dadam. The advanced information management prototype. In S. Abiteboul, P. C. Fischer, and H. J. Schek, editors, *Lecture Notes in Computer Science*, pages 3–26. Springer-Verlag, 1989.

[PD96] J. Patel and D. DeWitt. Partition based spatial merge join. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Montreal,Quebec, May 1996.

[PT86] P. Pistor and R. Traunmueller. A database language for sets, lists and tables. *Information Systems*, 11(4):323–336, 1986.

[PYK+97] J. Patel, J. Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N. Hall, K. Ramasamy, R. Lueder, C. Ellman, J. Kupsch, S. Guo, J. Larson, D. DeWitt, and J. Naughton. Building a scalable geo spatial DBMS: Technology, implementation and evaluation. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Tucson, Arizona, May 1997.

[RK87] M. A. Roth and H. F. Korth. The design of ¬1NF relational databases into nested normal form. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 143–159, 1987.

[RKB87] M. A. Roth, H. F. Korth, and D. S. Batory. SQL/NF: A query language for ¬1NF relational databases. *Information Systems*, 12(1):99–114, 1987.

[SAB+89] M. Scholl, S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, and A. Verroust. VERSO: A database machine based on nested relations. In S. Abiteboul, P. C. Fischer, and H. J. Schek, editors, *Lecture Notes in Computer Science*, pages 27–49. Springer-Verlag, 1989.

[SC90] E. J. Shekita and M. J. Carey. A performance evaluation of pointer based joins. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 300–311, 1990.

[Sch86] M. H. Scholl. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *Proceedings of the International Conference on Database Theory*, pages 380–396, 1986.

[Shi81] D. W. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1), March 1981.

[SHT$^+$99] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Scotland, 1999.

[SK91] M. Stonebraker and G. Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

[SPL96] P. Seshadri, H. Pirahesh, and T.Y. C. Leung. Complex query decorrelation. In *Proceedings of IEEE Conference on Data Engineering(ICDE)*, pages 450–458, 1996.

[SPSW90] H. J. Schek, H. B. Paul, M. H. Scholl, and G. Weikum. The DASDBS project: Objectives, experiences, and future prospects. *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[SS89] H. J. Schek and M. H. Scholl. The two roles of nested relations in the DASDBS project. In S. Abiteboul, P. C. Fischer, and H. J. Schek, editors, *Lecture Notes in Computer Science*, pages 50–68. Springer-Verlag, 1989.

[Sto87] M. Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the International Conference on Very Large Databases*, pages 289–300, Kyoto, Japan, 1987.

[Sto96] M. Stonebraker. *Object-relational DBMS: The Next Great Wave*. Morgan Kaufmann, 1996.

[SWH83] E. Sciore, D. S. Warren, and P. B. Henderson. A relational model of operating system environments. Technical Report 83/060, Department of Computer

Science, State University of New York at Stony Brook, Stonybrook, NY, 1983.

[TRSB93] W. B. Teeuw, C. Rich, M. H. Scholl, and H. M. Blanken. An evaluation of physical disk I/Os for complex object processing. In *Proceedings of IEEE Conference on Data Engineering(ICDE)*, pages 363–371, 1993.

[Val87] P. Valduriez. Join indices. *ACM Theory of Database Systems (TODS)*, 12(2), 1987.

[VKC86] P. Valduriez, S. N. Khoshafian, and G. Copeland. Implementation techniques of complex objects. In *Proceedings of the International Conference on Very Large Databases*, Kyoto, Japan, 1986.

[Zan83] C. Zaniolo. The database language GEM. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 207–218, 1983.

[ZM90] S. B. Zdonik and D. Maier. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann Publishers, Inc, 1990.

[ZMR98] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, December 1998.