Multi-Attribute Partitioning Methods for Associative Database Systems

by

Harold A. Stern

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Master of Science

and

Bachelor of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1989

© Harold A. Stern, 1989

The author hereby grants to MIT permission to reproduce and to distribute copies of this thesis document in whole or in part.

Signature of Author ..............................................................................................................

Department of Electrical Engineering and Computer Science

May 19, 1989

Certified by .................................................. Robert M. Mattheyses

GE Corporate Research & Development Center

Thesis Supervisor

Certified by .................................................. Stuart E. Madnick

Sloan School of Management

Thesis Supervisor

Accepted by .................................................. Arthur C. Smith

Chairman, Departmental Committee on Graduate Students
Multi-Attribute Partitioning Methods for Associative Database Systems

by

Harold A. Stern

Submitted to the Department of Electrical Engineering and Computer Science on May 23, 1989, in partial fulfillment of the requirements for the degrees of Master of Science and Bachelor of Science

Abstract

Data space partitioning can be used to greatly improve the performance of an associative hardware in the execution of relational database operations. When manipulating sets of data that are much larger than the memory capacity of a parallel architecture, partitioning can reduce both the volume of data that must be looked at and the number of operations that must be performed on each object examined.

A multi-attribute partitioning scheme is described for a database management system to be implemented on the Connection Machine. Task decompositioning algorithms which can take advantage of the partitioning will be presented. Methods for rigorously benchmarking this and other database systems will also be discussed.

Keywords: Database machines, database management, data structures, parallel algorithms, data-level parallelism.

Thesis Supervisor: Robert M. Mattheyses
Title: GE Corporate Research & Development Center

Thesis Supervisor: Stuart E. Madnick
Title: Sloan School of Management
Acknowledgements

I would like offer thanks to my advisors, Bob Mattheyses and Stuart Madnick, for all their help; without them, this thesis would not have been possible. And to Marilyn Pierce, who kept letting me turn in forms late.

To Carl Hewitt, who supported me these past three semesters, and the rest of the Message Passing Semantics Group for their help with Acore and Commentary. To Jeff, Paul, and Saadia, who kept me company all those late nights down the stretch.

To Mike, Marie, and everyone else at The Tech, who did their best to keep me from getting this finished.

And to Mom, Dad, Leslie, Stuart, and Lori — without your love and understanding I never could have survived.

It’s been fun.
## Contents

1 Introduction .......................... 9
  1.1 Thesis Goals .......................... 10
  1.2 Outline .......................... 11

2 Concepts ................................ 14
  2.1 Relational Databases .................. 14
      2.1.1 Relational Operators .......... 16
  2.2 The Connection Machine ............ 19
      2.2.1 Virtual Processing Elements .. 20
      2.2.2 The Stripe Data Structure .... 21
      2.2.3 Representing Relations with Stripes 22

3 Database Machines ................. 26
  3.1 Associative Processors ............ 26
      3.1.1 CASSM ........................ 27
      3.1.2 RAP .......................... 28
      3.1.3 VERSO ....................... 29
  3.2 MIMD Machines .................... 29
      3.2.1 DBC .......................... 30
      3.2.2 DIRECT ....................... 30
      3.2.3 GAMMA ....................... 31
      3.2.4 SABRE ....................... 32
      3.2.5 GRACE ....................... 33
  3.3 The Connection Machine .......... 33

4 Parallel Internal Algorithms .... 35
  4.1 Sorting Methods ................. 38
      4.1.1 Sorting Networks .......... 38
      4.1.2 Bitonic Sorting .......... 40
  4.2 Unary Relational Operators .... 42
List of Figures

2-1 A sample relation containing information on customers and their accounts. ........................................ 15
2-2 Projection of a Customers-Accounts relation on the Customer attribute. ................................. 18
2-3 The union of two Customer-Accounts relations (right). ................................................................. 19
2-4 Stripes on the Connection Machine. ................................................................................................. 23
2-5 Relations are represented as sets of stripes. ....................................................................................... 24

4-1 Comparators and inverter comparators. ............................................................................................. 39
4-2 A network which sorts a bitonic sequence. ......................................................................................... 41
4-3 A bitonic sorting network .................................................................................................................. 43
4-4 The internal join of two relations. ....................................................................................................... 48

6-1 A relation partitioned on two attributes. .............................................................................................. 62
6-2 An exact match query, retrieving partition F. ..................................................................................... 62
6-3 A file partitioned into a k-d tree. ......................................................................................................... 68
6-4 A file partitioned with Dynamic Multipaging. .................................................................................... 68
6-5 A Grid file directory. The darker partitions represent physical page boundaries. ............................ 71
6-6 The same set of points as in the multipaging example, but here, no partitions are split unless they are overflowing. ................................................................................................................. 71

7-1 RMAP vs. DYOP with even distribution of tuples. ........................................................................... 89
7-2 RMAP is far superior to hash-based methods when there is an uneven distribution of tuples. .... 90
7-3 Load factors observed for insertion with mean of 75,000 and deviation of 10,000. ....................... 91
7-4 An RMAP directory (and the corresponding tuples) for a relation with 3 partitions and 9 tuples on a four-processor machine. ......................................................................................... 91
7-5 Multi-attribute partitioning in action. .................................................................................................. 92
7-6 Compatible partitions for merging. ..................................................................................................... 93
7-7 Find-nth in action. ................................................................................................................................ 93
7-8 A partial match query, retrieving partitions C, F, and H. ............................................................... 94
Chapter 1

Introduction

In the past, the primary uses of high-performance parallel architectures have been computational, as meteorologists and cryptologists are among the few who have been able to justify the purchase of expensive vector processors like a Cray.

But it has been clear for some years that the dominant role of computers has been that of information processing, rather than number crunching. Hsiao and Madnick [HM77] propose an “information utility” as an alternative measure of the power of computer systems. Rather than looking at the number of gigaflops an a computer can achieve, the information utility is a measure of an architecture’s ability to store, retrieve, and manage information rapidly and efficiently.

It has often been said that the vast majority of all computing time is devoted to two things: searching and sorting. The problem is that most computers are not very good at either of these. Linear searching can be a very expensive procedure when carried out on collections of records that measure in the gigabytes, not to mention the difficulty of sort such a database: The cost of modifying conventional computers to meet the performance requirements of modern-day information processing systems is prohibitive; in order to improve the performance of database systems on such vital tasks, it is necessary to explore the possibility of utilizing specialized, parallel hardware to meet these goals.

Many researchers have developed experimental architectures that can perform some sort of parallel “associative” search strategy. Often such an architecture will consist of a central processor, usually a conventional computer, with a “back end
This back end, which is entrusted with performing many of the lower-level functions of a database management system, is comprised of a customized processor or configuration of processors, together with some special-purpose hardware capable of performing simple filtering operations.

For the most part, the potential envisioned from this application of parallel architectures has not been realized in practice. This has largely been the result of the infamous "I/O Bottleneck" — the fact that with large databases, the amount of data that must be manipulated far exceeds that combined capacities of these back-end processors. Any potential savings one might gain from the use of specialized database machines are overshadowed by the overhead of transferring data into and out of secondary storage. The cost of paging, then, becomes the limiting factor in the performance of a parallel architecture.

One factor which has compounded this problem over the years has been a failure on the part of some of these designers to realize that the associative power of specialized hardware alone is not an acceptable substitute for indexing. Conventional database systems have always relied on indices to achieve acceptable performance; early designers of parallel databases falsely believed that their concurrent hardware would eliminate the need for these auxiliary data structures. But when databases grow are much larger than the capacity of the hardware, as they are in any realistic application, associativity alone cannot be used to avoid searching large portions of the database. The success of any database storage structure, then, is dependent not only upon its brute-force strength but upon its ability to partition the database and eliminate records from consideration without actually loading them into the backend hardware.

1.1 Thesis Goals

This thesis will attempt to answer the question of how to best accomplish that partitioning. There are two disparate bodies of knowledge that must be brought to bear on
the issue: data structures for efficient retrieval and parallel algorithms for relational database operations. Although many methods exist for organizing data so that it can be rapidly accessed [Knu73, Ben75, LW77, BF79, LW80, Rob81, Lar82, MO82, Ous83, NHS84, Ozk85, GG86], and much effort has been expended exploring parallel relational algorithms [Slo70, Bab79, CLW73, Su79, DB73, OSS75, Sch79, BHK79, Bor82, Bit83b, BD83a, Bit84, VG84, FKT86, DeW86], surprisingly few of those works have attempted a serious unification of the two related themes.

This thesis will try to remedy this. First, the potential of the Connection Machine (CM) will be explored as a massively parallel backend for a database system. Concurrent algorithms for relational operations will be examined, and appropriate methods for implementing them on our architecture will be devised. Here we extend the work of [Chr84, Ton88], who devised algorithms for small relational databases that could be contained within the CM's main processor memory. By considering tables that are far too large to be fully present on the machine, the issue of overcoming the I/O bottleneck becomes a problem that must be dealt with.

To overcome the I/O problem, we must turn our attention to the wide range of data structures mentioned above that have been proposed for performing data storage and retrieval. Although the performance of the majority of these structures is well-understood for the problem of simple selections, the question of how to incorporate these structures into a full-scale relational database system needs to be better addressed. The usefulness of these methods for "expensive" relational operators such as projection and join needs to be evaluated. The characteristics that make a storage structure good for use in relational databases must be made explicit, and algorithms for performing these operators on data that has been partitioned must be developed.

1.2 Outline

Chapter 2: Concepts This chapter will present a brief introduction to the hardware and programming of the Connection Machine. A simplified model of data storage
and I/O will be demonstrated. Relational databases — and how we might represent them on a massively parallel architecture — will be shown as well.

Chapter 3: Database Machines  The notion of using a specialized, parallel processor as a back-end to improve the performance of a relational database system has been with us for a long time. Some efforts have failed miserably, while others are still alive and well. The strengths and weaknesses of these projects will be discussed, as well as their relevance to the problem at hand.

Chapter 4: Parallel Algorithms for Small Relational Databz Many algorithms have been proposed for performing relational operations on massively parallel architectures. Most of these assume data sets small enough to assign a single item to each processor. These “toy” algorithms will be useful as the building blocks for operators that function on real-sized data sets.

Chapter 5: External Relational Algorithms  The assumption that relations fit into the number of available processors will be broken in this chapter. I/O costs rather than CPU execution cycles, will become the major performance issue in an analysis of a variety of different algorithms for supporting relational operators on large databases.

Chapter 6: Multi-Attribute Partitioning  In order to limit the effects of I/O on our algorithms, this chapter will feature a discussion of partitioning methods. With intelligent partitioning schemes, relational operators can determine that portions of our database do not need to be examined in answering a query without having to incur the expense of loading them into the back end processor. Several different methodologies will be surveyed, and their strengths and weaknesses as choices for the Connection Machine will be assessed.
Chapter 7: RMAP Partitioning and Relational Algorithms  Algorithms for implementing a partitioning method presented in the preceding chapter will be developed here. Methods for partitioning data alone is not enough to guarantee efficient performance. In addition, relational operators that are both aware of the pre-processing that is inherent in the structuring of the data and are capable of exploiting that partitioning are needed. Algorithms to implement these “intelligent operators” will be shown as well.

Chapter 8: Benchmarking  This chapter will discuss issues involved in quantitatively measuring the performance of a relational database system. The need for a comprehensive generator which will produce random data satisfying a widely ranging number of parameters will be demonstrated, as will a system for accomplishing that goal.

Chapter 9: Conclusions  This chapter will feature an assessment of the performance measures, as well as directions for further research.

Appendix A: Implementation Details  The major data structures and interfaces to those structures will be described.
Chapter 2

Concepts

This chapter will present the major concepts to be discussed in the thesis. These include:

1. Relation database management systems.
2. The architecture of the Connection Machine.
3. The stripe model of data storage.

2.1 Relational Databases

This description of relational databases is taken mainly from [Dat86].

**Relational Database** A *relational database* is a collection of information that can be treated as a collection of tables. A *relation* would then be a single table. A sample relation is shown in figure 2-1. The only operations that users will be allowed to perform on these relations are those which will yield more relations.

**Tuple** The rows of a relation are called *tuples*. These tuples are often referred to as *records*. Each record is a *d*-dimensional tuple of values \( k = \{k_0, k_1, ..., k_{d-1}\} \), where \( d \) is the *dimensionality*, or *degree*, of the relation. \{Fassler, 10\}, \{Garrison, 12\}, \{Becker, 3\}, \{Zedeck, 1\}, and \{Huang, 19\} are the tuples in the above relation. The *cardinality* of a relation is equal to the number of tuples. Although
we represent a relation as a table of tuples, the ordering of tuples within a relation does not matter. Also, there cannot be duplicate tuples in the relation.

**Attribute** The columns of a relation are composed of pieces of data that share common characteristics. The attributes in the above relation are Customers and Account-IDs. Attributes are sometimes called fields. The ordering of attributes within a relation should not matter either.

**Domain** Each attribute $k_i$ may take its value from some limited set of possibilities $D_i$. The set of acceptable values that the tuples of a relation may have for a given attribute is called the domain of that attribute. The domain of Account-IDs might be the set of positive integers less than 10,000.

**Primary Key** The primary key of a relation is the set of attributes in the relation such that no two tuples contain the same value for all of the attributes. A primary key may be composed of a single attribute or many attributes. Every relation has a primary key; since there can be no duplicate tuples in a relation, we can always designate the entire set of attributes as the primary key.

**Secondary Attribute** An attribute which could assume the same value in a number of records. The secondary attributes of a relation are all the attributes which are not included in the primary key of that relation.

**Foreign Key** Attributes which have the same domain as the primary key of some other relation are called foreign keys. Foreign keys are most frequently used to link the records in one relation with the records in another relation.

<table>
<thead>
<tr>
<th>Customers</th>
<th>Account-ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fassler</td>
<td>10</td>
</tr>
<tr>
<td>Garrison</td>
<td>12</td>
</tr>
<tr>
<td>Becker</td>
<td>3</td>
</tr>
<tr>
<td>Zedeck</td>
<td>1</td>
</tr>
<tr>
<td>Huang</td>
<td>19</td>
</tr>
</tbody>
</table>

Figure 2-1: A sample relation containing information on customers and their accounts.
Account-ID might be a foreign key into a relation containing more information about the particular accounts. A foreign key does not have to be a secondary attribute; in the above example, Account-ID is part of the primary key of the relation.

### 2.1.1 Relational Operators

If we wish to maintain the relational model of data, we must restrict the ways in which users can manipulate the database. There are only a limited number of operations that can be performed on relations. These relational operators are:

<table>
<thead>
<tr>
<th>Insertion</th>
<th>Projection</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>Union</td>
<td>Cartesian Product</td>
</tr>
<tr>
<td>Deletion</td>
<td>Intersection</td>
<td>Join</td>
</tr>
</tbody>
</table>

**Insertion** When inserting a new tuple into a database, it is necessary to ensure that the tuple does not already exist in the relation.

**Selection** Selection is a method for constructing a horizontal subset of a relation; that is, the subset of tuples within a relation for which a user-specified predicate is satisfied. We will restrict the discussion to *boolean queries*. A boolean query is one that can be expressed as the conjunction of constraints on attribute values: either a single value or a range of values may be specified for each attribute. The database system must return all tuples satisfying the query.

If the query is allowed to specify conditions for more than one key in a given query, the search performed by the system is called *associative*. There are several types of associative queries we will be interested in. Although the distinction is not important to the relational model, it will have a role in a later discussion of data structures.

**Exact Match** The user specifies a single value for each of the attributes in a relation.

All that is left for the system is to verify whether or not the tuple the user
described is present in the relation. *Does Garrison share account-number 12?* would be an exact-match query that would fail in the relation shown above.

**Partial Match** The user specifies a single value for some subset of the attributes. The system must retrieve all tuples that have the specified values for the proper attributes. *Give me all users who share account-number 12* would be a partial-match query that should yield a single tuple: \{Garrison, 12\}.

**Region Query** The most general type of query, in which the user specifies a range of values, i.e. an interval \([L_i, U_i]\) for each of the attributes. We specify a range of values for \(d\) keys, and all records that have every value in the range are reported as the answers. *Give me all users who share accounts 10-21* would be a range query that should yield \{Fassler, 10\}, \{Garrison, 12\}, and \{Huang, 19\}.

**Deletion** Like selection, the users present the delete operator with an abstract specification of the tuples they wish to act upon. In this case, however, the specified tuples are removed from the relation.

**Projection** Projecting a relation involves removing some attributes from its definition. Project constructs a vertical subset of a relation; that is, a subset obtained by selecting specified fields and eliminating others (and also eliminating duplicate tuples within the attributes selected). The set of fields that are to be kept is called the projection domain. In figure 2.1.1, the Customer-Accounts relation on the left is projected on the Customer attribute, and the result is shown at the right.

**Union** The union of two relations, \(R\) and \(S\), is the set of all tuples belonging to either \(R\) or \(S\) (or both). Note that the two relations must be of the same degree, and that the corresponding attributes of the two relations must be defined on the same domain.
Intersection  The intersection of two relations, \( R \) and \( S \), is the set of all tuples that belonging to both \( R \) and \( S \).

Difference  The difference between two relations, \( R \) and \( S \), is the set of all tuples that belong to \( R \) but not to \( S \). Note that the order in which the two relations are specified matters.

Cartesian Product  The cartesian product of two relations, \( R \) and \( S \), is the set of all tuples \( T = \{t_1, \ldots, t_n\} \) such that every \( T \) is the concatenation of a tuple from \( R \) and a tuple from \( S \).

Join  Join is an operator that combines two relations over a common set of attributes. The common set of attributes is called the join domain. Some selection criteria is used to determine whether a tuple of \( R \) with a value of \( r_i \) for its join attribute can be concatenated with a tuple from \( S \) with join attribute value \( s_j \). Two tuples that can be concatenated together are said to be compatible. The result of joining relation \( R \) on field \( R_i \) with relation \( S \) on field \( S_j \) is the set of all tuples \( T = \{t_1, \ldots, t_n\} \) such that every \( t_z \) is the concatenation of a tuple \( r \) and \( s \) such that \( r_i \) is compatible with \( s_j \).

- Equijoin  The most common form of join query is the equijoin, where the selection criteria is equality.

<table>
<thead>
<tr>
<th>Customer</th>
<th>Acct #</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fassler</td>
<td>10</td>
<td>Fassler</td>
</tr>
<tr>
<td>Garrison</td>
<td>12</td>
<td>Garrison</td>
</tr>
<tr>
<td>Fassler</td>
<td>19</td>
<td>Becker</td>
</tr>
<tr>
<td>Becker</td>
<td>3</td>
<td>Zedeck</td>
</tr>
<tr>
<td>Zedeck</td>
<td>1</td>
<td>Huang</td>
</tr>
<tr>
<td>Huang</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>Zedeck</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2-2: Projection of a Customers-Accounts relation on the Customer attribute.
Sorting Although sorting is technically not a relational operator, its function is not only useful in displaying the results of relational operators but is crucial for their performance. We will therefore consider efficient algorithms for sorting to be the cornerstone of a successful database system.

2.2 The Connection Machine

The Connection Machine (CM) [Hil85, Cor87] is a single-instruction, multiple-data (SIMD) machine with up to 64K processors. Each data processor in a CM-2 has 64K bits (8 kilobytes) of bit-addressable local memory; a fully configured machine, then, contains 512 megabytes of memory.

The Connection Machine is a data parallel computing system. Data parallel computing is a paradigm in which an individual processor is associated with each data element. This computing style exploits the natural computational parallelism inherent in many data-intensive problems. Parallel data structures are spread across the data processors, with a single element stored in each processor's memory. These data structures are called parallel variables, or *pvars*. Processing element (PE), then, is a term which can refer to a processor's data as well as the physical processor itself. This ties in well with the first of our stated objectives: that of rapid, parallel search strategies. Since the CM is an associative hardware, searching of 64K pieces of data can be accomplished in a single step.

<table>
<thead>
<tr>
<th>Customer</th>
<th>Acct #</th>
<th>Customer</th>
<th>Acct #</th>
<th>Customer</th>
<th>Acct #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fassler</td>
<td>10</td>
<td>Coppola</td>
<td>5</td>
<td>Fassler</td>
<td>10</td>
</tr>
<tr>
<td>Garrison</td>
<td>12</td>
<td>Huang</td>
<td>19</td>
<td>Garrison</td>
<td>12</td>
</tr>
<tr>
<td>Becker</td>
<td>3</td>
<td>Labendz</td>
<td>19</td>
<td>Becker</td>
<td>3</td>
</tr>
<tr>
<td>Zedeck</td>
<td>1</td>
<td>Becker</td>
<td>3</td>
<td>Zedeck</td>
<td>1</td>
</tr>
<tr>
<td>Huang</td>
<td>19</td>
<td>Shandler</td>
<td>1</td>
<td>Huang</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Coppola</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Labendz</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Shandler</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2-3: The union of two Customer-Accounts relations (right).
High-speed data transfers between peripheral devices and CM memory take place through the CM I/O system. All processors, in parallel, pass data to and from I/O Buffers. The data is then moved between the buffers and the peripheral devices.

Each CM-2 processor chip contains one router node, which serves the 16 data processors on the chip. The network that connects the routers is in the form of a hypercube, which in a fully configured CM is 12-dimensional. A message sent from one processor to another travels from one router node to another until it reaches the chip containing the destination processor (unless something goes wrong, like congestion in the network). This ties in well with the second requirement the information utility imposes upon us: efficient sorting of large collections of data, aided by the hypercube network which allows us to execute in parallel thousands of comparisons between processing elements.

2.2.1 Virtual Processing Elements

As stated above, we are not just concerned with relations measured in megabytes; we must be able to accommodate gigabytes. At first glance, we are limited by the 512M capacity of the CM's main memory. But the situation is worse than that. If we wish to maintain this data-parallel model of assigning a single tuple to a single processor, we are also limited to a maximum of 64K data elements. The Virtual Processor facility [Hil85, Chr84] enables the machine to simulate a CM with more processors. However, this is more of a temporary patch than a real solution, for several reasons.

- Virtual processors are implemented by allocating a portion of a processing element's memory to each virtual processing element (VPE) it must simulate. A CM-2, with 64K processors with 8K of storage, can simulate a machine with 128K processors, each with 4K of storage. We cannot slice the memory of the CM into infinitesimally small pieces. The maximum ratio of VPEs to PEs is 16; greater than 16 and there is no guarantee that a VPE will have sufficient room to compute anything. So, rather than solving our problem of storing
large databases, this facility merely increases by a factor of 16 the capacity of
the machine.

• Users cannot dynamically set the number of virtual processors according to the
particular data to be operated on. Every time the user wishes to use a different
number of VPEs, it is necessary to cold boot the system. Not only is this a very
inefficient method for utilizing the machine, it makes it impossible to operate
on two virtual pvars of different sizes at the same time.

• There is also an efficiency issue to consider. Using virtual processors also slows
down the machine in a manner that is proportional to the ratio of the VPEs
to PEs. Essentially, the PEs have to execute a command once for each VPE
it represents. A user cannot specify that a query should be executed on only
the first 64K virtual processors and not the remaining 64K. A CM with 128K
VPEs, then, will not only appear to have half as much memory per processor,
it will also appear to run twice as slow.

2.2.2 The Stripe Data Structure

An alternative is to use "stripes" as the basic structure on which to support relations
on the Connection Machine. A stripe is a string of contiguous bits that are stored
in the same memory locations of each processor. A processor can store any valid
data object in the stripe; different processors may of course have different values,
or different types of values. A stripe is somewhat analogous to a vector, where the
elements in a vector are analogous to the contents of a stripe in individual processors.

The CM-2 stores its data in a collection of disks referred to as the DataVault. The
data in a stripe residing in secondary-storage cannot be accessed or modified; only

1Rumor has it that future releases will allow some measure of this ability, although the potential
flexibility is unknown at the present

2To be exact, it is possible to specify a currently-selected-set of processors which is comprised
solely of the first 64K virtual processors. However, the query will still be executed on the second
block of 64K VPEs. It just won't have any effect.
stripes residing in main-memory can be manipulated. When reading from or writing to the DataVault, the user can specify the number of bits to be moved in the transfer; these bits could correspond to a single stripe or multiple contiguous stripes.

We will consider stripes to be the basic unit of data transfer between the outside world, the Connection Machine, and the secondary storage in the form of the DataVault; therefore, we will not take into account the situation where a single I/O transfer moves multiple stripes into or out of main memory. In 2-4, we show a model of the Connection Machine and its secondary storage. Stripes are represented as dark bands which stretch across the processors, filling up the same locations in all the PEs. Note that stripes stored off-line are represented the same as stripes in main memory; we are not concerned with the actual mechanics of secondary storage.

2.2.3 Representing Relations with Stripes

As mentioned above, stripes will be the data structures used to manipulate relations on the CM. There are several alternative mechanisms, but the conceptually simplest is as follows:

- We define a data structure called a bucket which contains a set of tuples. The number of tuples in the bucket must be less than or equal to the number of processors. A relation is defined as a collection of buckets.

- Buckets are composed of a set of stripes. Each stripe corresponds to an attribute; the values all the tuples contained in the bucket projected onto that attribute are stored in the stripe. A tuple can be defined as the set of values for a given processor of all the stripes in a bucket. In 2-5, we show such a representation.

A stripe is essentially a pvar that chooses not to make use of the virtual processor facility. Although the CM can execute an operation on only one stripe at a time, there is no requirement that it execute an operation on all stripes, unlike with VPEs. Also,
Figure 2-4: Stripes on the Connection Machine.
Figure 2-5: Relations are represented as sets of stripes.
the number of stripes present in CM memory can potentially grow to be orders of magnitude greater than the maximum VPE:PE ratio. To take an extreme, we could have 4K 1-bit stripes on our machine, while there is the constraint on the VPE:PE ratio of 16.

In summary, we see that the Connection Machine brings us far along the way to meeting our objectives. It provides us with the means for searching tens of thousands of records simultaneously, as well as a mechanism for rapidly sorting those items. But there are difficult issues which arise when considering the effect of a database which is not only larger than the number of available processors, but also larger than the memory capacity of the CM. Because I/O is such an expensive operation, unless we have some means of limiting the number of stripes that must be examined in the answering of a given query, and also limiting the number of comparisons that must be made between records in different stripes, then we are little better off than the users of conventional, uniprocessor database systems.
Chapter 3

Database Machines

Database Machines — specialized processors designed to be used as high-performance backends to relational database systems — have been with us a long time. The evolution of these features has gone through several phases:

- In the late 1960s and early 1970s, associative processors [Su79] which were capable of accessing data by content and performing searches in parallel were in vogue. These systems were built on special-purpose hardware that allowed simple queries to be executed on relations as the data was being read off of the disk.

- By the end of the 1970s, dissatisfaction with the performance of these associative processors led to the introduction of a number of multiprocessor systems for relational databases. The degree of parallelism available varied widely among the systems, as some only possessed a few processor units.

- Today, the introduction of massively parallel architectures such as the Connection Machine has brought associative processing back in fashion.

3.1 Associative Processors

When Slotnick [Slo70] installed processor logic on the read/write heads of fixed-head rotating storage devices, the first database machine was born. These "logic-per-track" devices allowed database systems to execute selection operations "on-the-fly", as data
was input off the secondary storage.

These devices were associative processors: hardware capable of accessing data by content rather than by its physical location on the disk, and able to perform searches in parallel. By filtering out data that was not needed in the answering of a query before the tuples waste valuable I/O bandwidth, these devices helped alleviate the bottleneck between the secondary storage and the main memory of the query processor.

The performance enhancement afforded by these associative systems was obviously most apparent when searching for a particular data item. The benefits were not just limited to retrieval queries: insertions, deletions, and all other queries that used selection to operate on only a segment of the database took advantage of the filtering properties of associative processors. But despite the success at meeting the primary design goal of efficient execution of the selection operator, these systems fared less well at the "expensive" relational operators, such as projection and join. One attempt to remedy this situation, which met with rather limited success, was made in the Content Addressable File Store project [Bab79]. CAFS used a "joinability filter" to sieve out tuples that had no compatible tuples in the relation to be joined with.

### 3.1.1 CASSM

The first complete database machine design was the Content Addressed Segment Sequential Memory project at the University of Florida in the early 1970s [CLW73]. CASSM utilized fixed-head disks with cellular logic. These cells consisted of processor logic, a pair of read/write heads, and a memory track on a disk. A single controller was used to distribute instructions to the cellular processors and to communicate with the host, making this a SIMD device.

Data in CASSM was stored in `<attribute, value>` ordered pairs. All the data items of a record were stored in a physically contiguous segment of the disk, together with record and relation identifiers. Also heavily used were `mark bits` which indicated
whether a given tuple was still a candidate for a given query.

To select an item of data in CASSM required three revolutions of the disk. On the first time around, the processors would mark all the tuples that belonged to the proper relation. On the second pass the cells would mark the tuples that met the selection criteria, and finally on the third revolution the tuples would be collected and output.

3.1.2 RAP

The Relational Associative Processor [OSS75] emerged out of the University of Toronto in the late 1970s. It bore many similarities to CASSM: it was based on cellular-associative, head-per-track disks, and utilized mark bits for the execution of queries. Again, there was a linear controller that was used to perform operations such as aggregates, making this another SIMD machine. One difference was that the data was organized along a simpler structure, in that only tuples from the same relation could be stored on the same track.

In RAP version 1, the designers acknowledged the fact that the purely associative fixed-head disk model had serious shortcomings when it came to performing queries that required comparisons between tuples such as a join. Processing elements were physically linked together to aid in the processing of these inter-relational operators. The difficulty in achieving synchronization between the different processor proved too great, however, as the interconnections were removed from the next release of the system.

The need for any means of partitioning was not acknowledged, as the authors wrote in [Sch79]: "The key to RAP's performance ... is parallel processing. Parallel processing eliminates the need for indices, such as inverted lists or B-trees, for fast retrieval."
3.1.3 VERSO

One interesting extension of cellular-logic technology which took the philosophy to an extreme was the VERSO project that emerged from INRIA in France [Ban83]. The premise behind VERSO’s design seemed to be that since it is difficult to execute a join using associative processors, they shouldn’t be used for doing them. The architecture used associative hardware to perform on-the-fly selection with inexpensive VLSI filters. Binary operations such as join could only be accomplished if the relation was sorted by other hardware.

3.2 MIMD Machines

Despite the proliferation of associative database projects, they all shared some serious drawbacks. The biggest problem faced was that of I/O bandwidth and hardware cost. Because these early devices required hardware which was usually very expensive, the inability to configure them with adequate resources made them small and slow. Because the associative processors were so small, the I/O problem was exacerbated as data could not be moved in large enough quantities to get operated on. The time to load data into most architectures was orders of magnitude greater than the time to search the array-resident data, eliminating (or reducing) the performance advantages [BO79].

Boral and DeWitt, among the designers of the multiprocessor systems DIRECT and GAMMA, believed that the current state of the art in hardware did not offer sufficient I/O capacity to be an effective part of a database system. They wrote that “trends in mass storage technology are making database machines that attempt to exploit a high degree of parallelism to enhance performance an idea whose time has passed [BD83b].”

In response, a number of database machines were developed which featured a much higher grain of parallelism. These multiprocessor-based systems ranged from
those which were highly concurrent, like DIRECT, to those which featured only a few, very powerful processors, like DBC.

3.2.1 DBC

Near the end of the 1970s, a group of researchers at Ohio State reacted to the infeasibility of head-per-track disks and devised the Data Base Computer [BHK79]. The DBC was composed of a few functionally specialized components, each of which was devoted to data storage, or directory information, or security enforcement. The Mass Memory Unit featured several moving-head disks ("processor-per-head devices, which could examine the entire cylinder of a moving head disk in a single revolution) with the ability to do readout in parallel.

The processors were organized into two loops. The Structure Loop, which maintained the organization of the database, was responsible for indexing, for determining authorizations, and for clustering records received for insertion into the database. These indices were used to limit the number of cylinders that must be searched for a given query. The Structure Memory, which was used to search the index in parallel, was comprised of content-addressable associative memory.

The moving-head disks were part of the Data Loop; they were connected to a number of query processor which could perform elementary search operations.

DBC offered two levels of clustering: a primary clustering attribute and a secondary clustering attribute. The system determined these parameters based on access patterns. It tried to store all records with the same value for the primary clustering attribute into as few cylinders as possible.

3.2.2 DIRECT

DIRECT [DeW79, Bor82] was a multiprocessor-based system dating back to the end of the 1970s whose outstanding feature was the crossbar which connected memory modules and query processors. These memory modules served as caches for the mass
storage devices they were connected to. Data was transferred in fixed-size pages, as relations were divided into blocks of 16 kilobytes. This relatively small page size was chosen to afford them the opportunity to have more memory modules present in the system.

When performing queries such as join, DIRECT used a “parallel nested loops” method. Query processors would load a page from one relation, and the memory modules would then broadcast the contents of a page from the second relation to all the processors in parallel. Because of the potential concurrency control issues, the designers of DIRECT foreswore the use of mark bits, choosing to make copies of relations rather than worrying whose mark bits were whose.

An additional difference between DIRECT and its predecessors was its ability to handle inter- as well as intra-query parallelism. Designed to function in a multi-user environment, DIRECT could achieve extra parallelism by having the query processors working on different queries at the same time. Priority could be changed dynamically as queries are allocated more or fewer processors.

Although the multiprocessor-based design was in part a reaction to the I/O problems suffered by associative architectures, the designers of DIRECT apparently had not fully learned their lesson. The initial implementation of DIRECT did not use indices or ordered relations, instead relying on parallel processing and simultaneous access to a single page by multiple query processors.

### 3.2.3 GAMMA

DIRECT's architects rectified some of their mistakes when they designed GAMMA [DeW86]. The GAMMA architecture consisted of 20 VAX 11/750 computers linked together, each possessing 2 megabytes of memory. Eight of those 20 were connected to 160-megabyte disk drives. GAMMA was not, however, a distributed database system. There was no local autonomy, as all query execution was centrally controlled.

One major design goal in GAMMA was to maximize the amount of work performed
locally, in order to reduce communications overhead. Relations were horizontally partitioned across all the disk drives in the system using a variety of methods. Indices could be used to maintain all of the following partitioned views:

- **Round-robin:** The default strategy for query results, this method simply called for the tuples in a relation to be spread out evenly across the disks.

- **Hashed:** Tuples are hashed on some key, and the hash value determined which disk the tuples should be stored on.

- **Range Partitioning:** The user could specify the regions of key values that should be kept on each disk. Alternatively, the user could specify that the system should attempt to maintain an even distribution of tuples while keeping them partially ordered on that key.

### 3.2.4 SABRE

SABRE, the latest database machine out of INRIA, featured a very low degree of parallelism, relying instead on specialized processors in conjunction with filtering [Che86, VG82, VG84, VV84]. The filters were not the traditional input selectors attached to read/write heads; recognizing the need for partitioning, the designers gave these filters the capability of intelligently assigning tuples to areas of secondary storage. This partitioning was both multi-attribute (on more than one key) and multi-level (tree-structured, with the possibility of different attributes serving as the discriminator at different levels).

In the other direction, when data is read off the disks, it is moved via a bus to cache memory. Processors designed specifically for join or sorting access data from the cache memory via an interconnection network. The query processor send read/write requests to the cache to get pages of the relation and to write back results. There is a parallel transfer of data between cache and join processors.
3.2.5 GRACE

One of the newest entrants into the already-crowded database machine field is the University of Tokyo's GRACE project [KTMo83, KTMo84, Fus85, FKT86]. Much higher parallelism is achieved than in the SABRE project, as GRACE resembles DIRECT in its parallel transfer of data from cache memory to query processors. The architecture features two ring buses connecting four specialized types of modules. The "staging ring" connects the disk modules to the memory modules, and the "processing ring" links the memory modules to the query processors. Control processors sit on both buses, monitoring the flow of data and issuing instructions.

GRACE uses on-the-fly hashing to carry out the expensive relational operators. In the staging phase, streams are generated from the disks and hashed by filter processors to a set of memory modules. A single hash bucket is distributed across all the memory modules. A single query processor will execute the relational operator on a given bucket, so it must extract its data from all of the cache processors. The processors' accesses are pipelined; that is, when processor $i$ is finished with memory module $j$ and moves on to memory module $j + 1$, then processor $i - 1$ (which was extracting data from memory module $j - 1$) will move on to $j$.

3.3 The Connection Machine

With the recent appearance on the market of a massively parallel associative processor in the Connection Machine, a number of research projects have examined the possibility of turning back to the pre-MIMD days of database machines.

Clearly, any platform for a relational database backend should possess some form of associative memory in order to be able to rapidly search a large quantity of data simultaneously. It should be able to rapidly sort records on a variety of different domains. And it should be able to perform I/O in parallel to help alleviate the bottleneck. The Connection Machine, with its tens of thousands of processors for
associative searching, and a multiply-connected routing network for rapid sorting, meets the specifications admirably. And it has one further advantage that few of the above multiprocessing systems do: it is already in production.

**Document Retrieval**  Stone [Sto87] examined the issue of identifying articles from keywords in a document retrieval system using the Reuters news service. The basic task was: given a keyword, how can you easily locate the files that contain that keyword?

Stone’s conclusions focused on the need for indexing in associative processors: he found that indices provided a major performance advantage over algorithms that did not use indexing, reducing data transfer so much that in many cases, a serial algorithm that used indexing was able to beat one running without indexing on the Connection Machine’s 64,000 processors. The limitation of this work, as with many studies performed on associative systems, was that it failed to address the traditionally difficult relational operators, focusing instead on what they are best at — pattern matching.

**Relational Databases on the CM**  Christman [Chr84] postulated that the Connection Machine would be well-suited for executing relational queries. This theory was fleshed out by Tong [Ton88], who implemented the methods presented in Christman’s thesis. These efforts focused solely on the case where there were more processors than the number of records in a relation, limiting their applicability.
Chapter 4

Parallel Internal Algorithms

Most analyses [Chr84, Ton88, BM87] of the usefulness of the Connection Machine as a relational database platform have limited themselves to the case where the number of tuples is less than the number of active processors. These analyses fail to accommodate the following two scenarios, for which solutions will be proposed in the next chapter:

- A relation is too large to fit into a single stripe, but small enough to be contained in the Connection Machine's main memory. As stated above, we will not simply use the virtual processor facility to hand-wave-away the issue. By using stripes, a database system can perform optimizations which are not available when virtual processors are in use.

- The relation is too large to be contained in the Connection Machine's main memory. In this case, not only do we have to consider the cost of executing an operator on multiple stripes, but we also must concern ourselves with the expense of loading these stripes into and out of the main memory.

Although we will be extending our system to perform under these circumstances, it is still useful to begin with these algorithms for executing relational operators on "toy" databases. Methods for use with single stripes can be useful as building blocks for algorithms which perform on larger relations. Although it is possible for the Connection Machine to operate on multiple stripes at a time, the potential for operation on multiple stripes simultaneously is limited both by the size of the machine and
the capacity of the programmer to develop more complex code. Therefore, relational
operators for large databases will have to be structured as follows:

- An *internal-operator* performs the operation on the tuples contained in a single
  bucket or a small set of buckets.

- A *merge-operator* takes the buckets which have already been operated on and
  combines them into runs of buckets which contain the same results as if the
  internal-operator had been applied to a single bucket of twice the size on a ma-
  chine with twice the processors. This merge-operator is applied to successively
  larger runs until the buckets appear the same as if the internal-operator had
  been run on a machine with an infinite number of processors.

We will discuss the internal operators in this chapter, deferring discussion of the
merge operators until the next one.

When the relations are small enough to fit into a single stripe, the relational
operators are simple to implement. Since every tuple is in main memory at the same
time, and can be examined simultaneously, selection can utilized the full power of
the associative architecture. The routing network enables us to rapidly perform the
tuple-comparison operations we need for sorting and other relational operators.

**Insertion and Deletion** When inserting tuples into a relation, the system must
keep track of the values in a stripe that are not considered part of a relation. In
figure 2-5, we see *nils* in the two stripes which correspond to the tail ends of the
Customer and the Account-Number attributes. To prevent these values from being
considered as real data, the system will maintain a one-bit attribute, * Included,* with
every bucket. This auxiliary attribute indicates for each processor whether the stripe
values of that processor comprise a tuple of the relation.

When inserting a tuple, it is sufficient to simply locate a non-full bucket (where
the capacity and number of tuples in each bucket is included in some data structure
in the host), locate a processor whose value for the included-stripe is nil, and insert
the tuple's attribute values into each of the proper stripes. The included stripe is set
to \( t \) for the processor into which the tuple was inserted. Deletion only requires that
we set the included-stripe for the processor(s) containing the deleted tuples(s) to \( \text{nil} \).

For reasons of efficiency, we will amortize the cost of performing our relational
operators by incurring some additional overhead on insertions and deletions. In par-
ticular, the tuples within a particular set of stripes will be kept in sort order on that
primary key. For example, if the value of tuple \( r_1 \)'s primary key is less than the value
of tuple \( r_2 \)'s primary key, and if \( r_1 \) and \( r_2 \) are contained in the same bucket, then
the processor containing the values of \( r_1 \) will have a lower address than the processor
containing \( r_2 \).

If the primary key is a compound one, the tuples will be sorted on the concatenation
of the attribute values. This invariant is maintained on insertion by broadcasting
the new tuple's value to all the processors. All processors whose values are greater
than the new record "shift" their values, by reading the tuple from the preceding
processor. Tuples obviously remain in sorted order in the face of deletions.

Because relations are stored in sorted order on the primary key fields, we can
easily guarantee that no duplicate tuples will be contained in the same bucket. Any
two tuples with the same primary key field (which, by definition, would be duplicate
tuples) would be stored in consecutive processors.

Of course, if the user should request that a relation be sorted on a non-key field,
then we have to modify this constraint in the following fashion:

- \textit{Relations which are sorted on a non-key field will be stored so that if two tuples}
  \textit{have the same value for the sort attribute, then they will be maintained in the}
  \textit{order of their primary key.}

In this case, the detection of duplicate tuples would require that the new tuple
value be broadcast to all of the processors for comparison. Any processors containing
tuples that match the one being inserted are deleted.
Selection  With an associative memory, selection on a relation contained in a single set of stripes is a trivial issue. For each attribute specified in the query, the query processor examines the stripe corresponding to that attribute to determine whether the tuple should be included in the new relation. The results of applying that sub-query to each of the \( k \) attributes specified in the query are \textit{and-ed} together, and the result is used as the included stripe for the new relation.

With the exception of cartesian product, all other internal relational operators will require a sorting algorithm for efficient performance. A digression into the domain of parallel internal sorting algorithms is therefore appropriate.

4.1 Sorting Methods

There are several parallel sorting algorithms which can be executed in \( \Theta(\log^2 p) \) steps on an associative architecture with \( p \) processors. One class of such sorting algorithms, is the class of network sorting algorithms [Bat68, Knu73]. As we shall see, these naturally map onto the Connection Machine structure [Chr84, Hil85, Ton88].

4.1.1 Sorting Networks

Network sorters utilize \textit{comparators}, which take as inputs two elements, \( x \) and \( y \), and outputs \( \min x, y \) on the upper output and \( \max x, y \) on the lower output. Data flows from left to right on horizontal lines (also called wires), and we draw comparators as vertical lines which connect two hollow circles. In figure 4-1, we also represent negative comparators, which are drawn with solid circles. Negative comparators output \( \max x, y \) on the upper output and \( \min x, y \) on the lower output. The time needed for a sorting network to compete is equal to the maximum number of comparisons made on any single wire, which can be viewed as the maximum sequentiality of the network.

For sequential architectures, these network sorters are too inefficient for practical
Figure 4-1: Comparators and inverter comparators.
use. They make the same comparisons between the same pieces of data no matter what the data looks like. But as a parallel algorithm, this non-adaptiveness is ideally suited to a SIMD architecture. The regular pattern of the comparisons coincides with the Connection Machine's broadcast of a single instruction to all the processors simultaneously. The comparison/exchange model of sorting networks resembles quite closely the message-passing performed by the router. This router provides an intuitive framework for making a set of comparisons between processors where the pattern of comparisons is independent of the particular data involved.

There are several variants of network sorting algorithms, but they all proceed in a similar fashion: a set of elements is distributed among the processors. The values in adjacent processors are compared, and exchanges are made where necessary to form runs of length two. Then these runs are combined into sorted runs of length four, and so on until the entire set is sorted. Parallel internal merge sorting takes $\Theta(\log(p))$ of these merge steps to complete. Sorting networks developed by Batcher [Bat68], which will be demonstrated below, are capable of merging two sorted runs together in $\Theta(\log(p))$ steps, giving us a performance of $\Theta(\log^2(p))$.

### 4.1.2 Bitonic Sorting

Internal bitonic sorting algorithms are based on the sorting networks. A *bitonic sequence* is a concatenation of two monotonic sequences, one of which is increasing and one of which is decreasing. Either one can come first; a bitonic sequence, then, is one which increases and then decreases or decreases and then increases. The following are two examples of bitonic sequences:

1 2 3 4 5 5 4 3 2 1 0 and 5 4 3 3 2 1 0 0 2 4 6 8 10

**Sorting a Bitonic Sequence** Given a bitonic sequence of length $n$, we can construct a sorting network that will merge it in $\log(n)$ steps, using the pattern depicted
Merging Sorted Sequences with Bitonic Networks  Given two sorted sequences of length $n/2$, we can merge them together to form a sorted sequence of length $n$ in $\log(n)$ steps, simply by inverting the second sequence and utilizing the bitonic sequence sorter shown above in 4-2. Instead of inverting the second sequence, we could just invert the comparisons made on the first level; we could compare processor 0 with processor $n-1$, 1 with $n-2$, and so on. This would accomplish the same thing, but it is important to note that a comparison of this form would take significantly longer to execute on a real machine. In the first case, all comparisons are made between processors whose addresses differ only by a single bit — that is, processors which are directly connected along some dimension by the routing network. The distance between processor 0 and $n-1$, on the other hand, is equal to the number of dimensions
in the configuration — a much longer path for the data to travel.

**Bitonic Sorting** Finally, the merging network developed above can be used as the building block for a sorter which sorts all input sequences, and not just those that happen to be bitonic. The sort proceeds in stages, as the merge networks are applied to longer and longer sequences of values. In the first stage, we simply compare the values in adjacent processors to form sorted sequences of length two. Note that some of the comparators must be inverters, since the mergers in stage 2 have to receive bitonic sequences, and not sorted sequences, from the mergers in stage 1.

In this second stage, we feed the \( n/2 \) sorted sequences of length 2 into either the top half or the bottom half of the \( n/4 \) bitonic mergers. This iterative rule is successively applied, until stage \( \log(n) \), where we merge the two sorted sequences of length \( n/2 \) into a single sorted sequence.

### 4.2 Unary Relational Operators

Unary relational operators are those which are carried out on a single relation. They include insertion, deletion, and selection (treated above), as well as projection.

**Projection** The difficult part of performing a projection is to remove the duplicate tuples that are created. The most straightforward algorithm sorts the tuples on some attribute, or set of attributes, which comprise a primary key for the result relation. Once the relation is sorted it is trivial to detect duplicates; they would be located in consecutive processors. Now if one of the attributes used in the projection is the primary key, and we have properly enforced the constraint that we will only insert tuples that do not already exist, then there is no need to do any further processing; there will be no duplicates. The system does not have to perform any additional computation to carry out a projection other than forming a new data structure in the front-end host.
Figure 4-3: A bitonic sorting network
If, on the other hand, the primary key is not totally included among the attributes in the new relation, then we must determine a new key on which to sort. Apart from any other information about the data, we must sort the tuples using the entire tuple as a key. After sorting on the superkey, each processor looks at the value of the tuple in the preceding processor, and if they are equal, removes itself from the relation. Execution of a projection in this case would require $\log^2(p)$ time to perform the sorting using the internal sorter described above and $\log(p)$ time to pack the values using the following method:

1. Set the list of active processors to be those processors still containing values.

2. Use the starlisp parallel enumeration function which returns a pvar such that the first active processor contains a 0, the second active processor contains a 1, and so on. This can be accomplished in $\log(p)$ time using the algorithm described in [Chr84].

3. Each active processor sends the value of its tuple to the processor whose address was generated in the enumeration. If there are $n$ active values, then, they will be contained in processors $\{0, 1, \ldots, n\}$.

### 4.3 Binary Relational Operators

In order to maintain the single-stripe model for our relational operators, we must further restrict the size of our relations to half the number of processors in the case of those operators which merge two relations to form a single output relation. These operators are union, intersection, and difference.

Later, the maximum size of input relations will have to shrink even further, down to the square-root of the number of processors for join and cartesian product; the maximum size of the output of these binary relational operators is the product of the sizes of the two input relations.
Union Like projection, the important issue involved in performing a union is the removal of tuples that are duplicated in the two sub-relations. Since there could be tuples present in relation $R$ that are also present in $S$, union is analogous to a projection where the primary key is not included among the projected attributes.

The two relations are merged together and sorted. Then each processor looks at the tuple stored in the next processor. If the value of the tuples in the two processors are equal, then the processor doing the looking deletes its tuple. We then enumerate the set of non-duplicates, and compact the set of tuples into the lowest-ordered processors. Note that we do not care which relation the tuples came from.

Intersection The intersection algorithm is similar to that of union, except that the system looks for duplicate tuples rather than eliminating them. Also different in this case is that we are interested in which relation a tuple came from; a tuple which (for some unknown reason) appeared twice in a bucket of $R$ would not be a candidate for membership in the intersection of $R$ and $S$ unless it also appeared in a bucket of $S$.

Instead of sorting on the primary key alone, it is necessary to modify the key field we will be sorting on with a bit indicating the source relation. Use a 0 for $R$ and a 1 for $S$, so that if a tuple from $R$ has the same primary key as a tuple from $S$, it will come first when they are merged and sorted. After sorting, each processor that has a tuple from $R$ looks at the value of the tuple stored in the next processor. If it is the same tuple, and is from $S$, then it is marked as belonging to the output. All tuples that are not marked as belonging to the output are deleted. The source relation bit is removed, and the tuples are again compacted into the low-ordered processors.

Difference This time, we are looking for tuples in $R$ that are not in $S$. As with intersection, the two sets of tuples are merged and sorted. Each processor $i$ that has a tuple from $R$ looks at the tuple in the next processor. If the tuple in processor $i + 1$ has a different value, then the tuple in processor $i$ is marked as belonging to the output. All tuples not so marked are deleted before compaction occurs.
Cartesian Product Unlike the binary operators discussed above, there is no need to sort before performing a cartesian product. To form the resultant relation \(|S|\) copies of \(R\) will be made, sending tuple \(R_i\) to processors:

\[
\{ |S|i, |S|i + 1, ..., |S|*(i + 1) - 1 \}
\]

\(|R|\) copies of each tuple in \(S\) will also be made, sending tuple \(S_j\) to processors:

\[
\{ j, j + |S|, + 2*|S|, ..., j + |R|*|S| }.
\]

Rather than having to go through \(|R| + |S|\) sends, we will perform all the sends in a few short steps. First we determine the size of the resultant relation, which is simply the product of the sizes of \(R\) and \(S\). Then each processor (with address \(a\)) which is to contain a tuple in the output grabs the \((\text{floor} \frac{a}{|S|})\)'th tuple from \(R\). Ignoring what is going on in the underlying routing network, we can accomplish this in a single step with a parallel read operation. Next, processor \(a\) grabs the \((\text{mod} a S)\)'th tuple from \(S\), again using a single parallel read operation. The concatenation of the stripes into a single relation can then be carried out in the main host.

4.3.1 Join Algorithms

There are several different algorithms available for performing the joins of two relations. The nested loops algorithm, where each tuple is compared to every other tuple, is virtually identical to the cartesian product method described above followed by a selection. For the general case of join, this is the only algorithm available. But in the common case of equijoin, the sort-merge offers significantly better performance.

Equijoin Algorithm The join attribute of each tuple is modified by concatenating it with a unique identifier. The most significant bit of this unique identifier will
indicating which relation the attribute belongs to (a 0 for \( R \) and a 1 for \( S \)). The least significant bits will indicate what processor the tuple was originally stored in.

The join attributes of the two relations are sorted. Since we cannot guarantee that the sum of the cardinalities of the two relations will be less than the number of processors, we must sort them individually and then merge them together. This will appear as follows:

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
A_0 A_1 a_0 a_1 B_0 B_1 C_0 c_0 c_1 c_2 D_0 D_1 d_0 d_1 F_0 g_0
```

where \( A_0 \) is the first tuple in \( R \) to have a join attribute value of \( A \), \( A_1 \) is the second, and \( a_0 \) is the first tuple in \( S \) to have that join attribute value.

Instead of having to calculate the cartesian product of the two relations, sorting reduces the task to the union of a set of cartesian products; we only have to compute the cross products of those sets of tuples that share the same values of the join attribute. This is done in a similar fashion to the way the cartesian product was done, except that the blocks of joinable tuples must be concerned with their offset.

Each processor \( i \) containing a tuple from \( R \) must make \( s[i] \) copies of it, where \( s[i] \) is the number of tuples from \( S \) with the same join attribute. Likewise, each processor \( j \) containing a tuple from \( S \) must make \( r[j] \) copies of it, where \( r[j] \) is the number of tuples from \( R \) with the same join attribute. The goal is the situation displayed in the bottom of figure 4.3.1, where each processor whose address is less than the output size contains a tuple from \( R \) and a tuples from \( S \).

To get there, we first distribute to each processor the number of tuples in both \( R \) and \( S \) with the same join attribute (the stripes labeled \( r \) and \( s \), respectively, in the above figure). We also distribute the relative ranking of each processor within the group of tuples from the same relation that share a join attribute (stripe \( c \)).
each processor is told how many members of the output are produced by tuples with a smaller join attribute. If output tuples are stored in order of their join attribute, this figure represents the offset at which processors whose tuples share a common join attribute can begin forming their output (stripe o).

In the figure, the first processor containing a tuple from $R$ (processor 0) makes 2 copies of $A_0$ and places them in output locations 0 and 1. Processor 1, places the two copies of $A_1$ in output locations 2 and 3. Processor 7 (the next one that contains a tuple from $R$ that has compatible members from $S$) takes up the next two spots with $C_1$. In general, processor $i$ containing a tuple from $R$ starts placing its $s[i]$ copies in processor $o + c \times s$ (the offset of the block of tuples with the same join attribute as $i$ plus the number of tuples that have already been placed down by tuples with the same join attribute) and finishes at processor $o + c \times s + (s - 1)$.

Tuples from $S$, on the other hand, must alternate to ensure the proper pairings arise. They are placed in cyclic order within the block of output tuples with the same join attribute. In the figure, processor 2 places $a_0$ in the first output location, and processor 3 follows with $a_1$. It is back to processor 2 again with $a_0$, and processor 3 completes the block with $a_1$ in location 4. In general, processors containing tuples from $S$ place them in locations $c + o$ (it's relative position in the sequence of tuples from $s$ with the same join attribute, offset by the number of tuples already output), location $c + o + s$ (the next time around), $c + o + 2 \times s$, and so on until $o + c + (r - 1) \times s$.

<table>
<thead>
<tr>
<th>name</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>$A_0$</td>
<td>$A_1$</td>
<td>$a_0$</td>
<td>$a_1$</td>
<td>$B_0$</td>
<td>$B_1$</td>
<td>$C_0$</td>
<td>$c_0$</td>
<td>$c_1$</td>
<td>$c_2$</td>
<td>$D_0$</td>
<td>$D_1$</td>
<td>$d_0$</td>
<td>$d_1$</td>
<td>$F_0$</td>
<td>$g_0$</td>
</tr>
<tr>
<td>r</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>s</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>o</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

*From the above, we get the following joined tuples:*

<table>
<thead>
<tr>
<th>$R'$</th>
<th>$A_0$</th>
<th>$A_0$</th>
<th>$A_1$</th>
<th>$A_1$</th>
<th>$C_0$</th>
<th>$C_0$</th>
<th>$C_0$</th>
<th>$D_0$</th>
<th>$D_0$</th>
<th>$D_1$</th>
<th>$D_0$</th>
<th>$D_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S'$</td>
<td>$a_0$</td>
<td>$a_1$</td>
<td>$a_0$</td>
<td>$a_1$</td>
<td>$c_0$</td>
<td>$c_0$</td>
<td>$c_1$</td>
<td>$c_2$</td>
<td>$d_0$</td>
<td>$d_0$</td>
<td>$d_0$</td>
<td>$d_1$</td>
</tr>
</tbody>
</table>

Figure 4-4: The internal join of two relations.
Chapter 5

External Algorithms

In the preceding section, we examined algorithms for relational operators which rely upon the assumption that the sum of the number of records in the input and output relations is less than the number of available processors. This assumption, which in real-world databases will never be met, severely limits the difficulty of developing algorithms for relational operators.

Unfortunately, physical reality places limitations upon these algorithms. Since the number of processors, although large, is finite, we may have more records in our relation than we do processors. If we cannot store our entire database in a single stripe then the above methods will not work without modification. And since there is only a finite amount of memory at each processor, the total combined memory of all of the processors may be less than the sizes of the relations we wish to work with. We cannot, then, assume that a whole relation can exist in Connection Machine main memory before processing begins, or at any given time during the processing.

5.1 External Merge Sorting

When we try to sort very large sets on the Connection Machine, we find that its architecture is not well suited for external sorting. Unlike a MIMD multiprocessor system, the Connection Machine can only operate on a single data set at a time. Our algorithms for external sorting, then, will very closely resemble those used on a conventional serial computer, except that where the conventional machine can only
make a single comparison on a single page of data at a time, the Connection Machine makes multiple comparisons from that same single stripe. A number of methods exist for performing external sorting [Knu73, Bit83b, Bit84], all of which are some variant or another of merge sorting.

We will need to make use of the following primitive operators:

Load(s)  Reads the stripe s out of secondary storage and loads it into the Connection Machine's main memory.

Store(s) Saves a stripe s out to secondary storage.

Sort(b, a)  The tuples contained in bucket b are all sorted, using the values of attribute a as a key. This is accomplished in the following steps:

1. For each processor that contains a tuple of the bucket, determine the rank of that processor's value of the key attribute. This can be done by forming a new key whose higher-ordered bits correspond to the attribute value and whose lower-ordered bits correspond to the address of the processor. This stripe is sorted using the methods detailed in the previous chapter. Then each processor that contains a tuple of the sorted stripe extracts the source-address bits, and sends its own address to the indicated processor.

2. For each stripe in b, have each processor that contains a value send that value to the processor whose address it received in the previous step.

The cost of the sorting in the first step is $\Theta(\log^2(p))$ cpu cycles. If there are k stripes in the bucket, then the cost of the second step will be $\Theta(k)$ i/o operations, one for each stripe that must be re-arranged.

Merge(b₁, b₂, a)  Given two buckets b₁ and b₂, such that b is sorted positively on attribute a and b₂ is sorted in reverse order on that same attribute, merge them together so that:
1. Every tuple contained in $b_1$ has a value for the key attribute less than that of every tuple contained in $b_2$.

2. Both $b_1$ and $b_2$ are sorted positively on the key attribute.

The merger proceeds in two phases; in the first phase, comparisons and exchanges are executed in parallel between corresponding stripe values in each processor. Having to make sequentially run through all the stripes in the buckets to accomplish this could cost $\Theta(k)$ i/o operations. At the end of this phase, all the tuples in the first bucket have key attribute values that are less than that of every tuple in the second bucket. At this point, we "clean up" the two buckets to put them back into sort order. This step, which corresponds to the final phase of a bitonic merge sorter, can be completed in $\log(p)$ cpu time and another $\Theta(k)$ i/o operations as each stripe is re-arranged.

Comparison Phase Each processor compares its value of $r$ to its value of $s$, storing the smaller value in $r$ and the larger value in $s$. The comparisons made are analogous to the ones that would be made when a machine with $2p$ processors was merging the two segments of length $p$. This step is sufficient to ensure the first condition: all values greater than or equal to the median will be stored in $s$. $T(n) = \Theta(1)$

Cleanup Phase 1 Let $d = \log(p)$ equals the number of dimensions on the machine. Let each processor compare its $r$ value with that of the processor whose address is equal to the address of the first processor with its $d^{th}$ bit flipped. Store the smaller value in the smaller-numbered processor. Repeat for all $d$. The comparisons are analogous to those made in the final phase of a bitonic merge of $p$ processors. This ensures the first half of the second condition, by sorting $r$. $T(n) = \Theta(d) = \Theta(\log(p))$

Cleanup Phase 2 Repeat cleanup phase 1 for $s$. This re Sorts $s$, and completes our goal. $T(n) = \Theta(d) = \Theta(\log(p))$

Merging of two runs of sorted stripes To merge two runs of length $l/2$ together (where $l = n/p$), we do the following:

1. Let first-half $= \{r_1, r_2, ..., r_{l/2}\}$ and second-half $= \{s_1, s_2, ..., s_{l/2}\}$.
2. For each pair \((r_i, s_i)\), merge the two stripes together using the above algorithm.

3. Call \textit{merge-runs} recursively on the first-half and the second-half.

\[
T(l) = 2T(l/2) + \Theta(\log(p)) \\
T(l) = \Theta(l \cdot \log(p))
\]

**Sorting a set of stripes**  Finally we are at the point where we can sort a set of \(l\) stripes. We have the following parameters:

\[
\begin{align*}
  n &= \text{the number of records in the database} \\
  p &= \text{the number of records in a stripe} \\
  l &= \text{the number of stripes in the database} = n/p \\
  io &= \text{the cost of reading and writing a stripe} \\
  s &= \text{the cost of internally sorting a stripe} = \Theta(\log^2(p)) \\
  m &= \text{the cost of merging two stripes} = \Theta(\log(p))
\end{align*}
\]

1. Sort each of the \(l\) stripes.

\[
T(l) = l \cdot io \cdot s.
\]

2. Call \textit{merge-stripes} on each pair.

\[
T(l) = l \cdot io \cdot m
\]

3. Call \textit{merge-runs} on each run of length two.

\[
T(l) = l/2 \cdot io \cdot 2 \cdot \log(p)
\]

4. Call \textit{merge-runs} on each run of length four.

\[
T(l) = l/4 \cdot io \cdot 4 \cdot \log(p)
\]

\[
\log l \quad \text{Call \textit{merge-runs} on the two runs of length \(l/2\).}
\]

\[
T(l) = 1 \cdot io \cdot l \cdot \log(p)
\]
\[ T(l) = \Theta(i_0 \ast (l \ast s + l \ast m + \log(l) \ast l \ast \log(p))) \]
\[ T(n) = \Theta(i_0 \ast n/p \ast \log(p)(\log(p) + 1 + \log(n/p))) \]
\[ T(n) = \Theta(i_0 \ast n/p \ast \log(n) \ast \log(p)) \]

5.2 Dynamically Constructing a Large Relation

When trying to maintain dynamically changing large relations, we are faced with the fact that there is no simple method for maintaining desirable characteristics of our data, such as the constraint that (in the absence of any explicit sort operation) tuples be stored in the order of their primary key attribute.

The natural mechanism for dynamically inserting records into a relation is to start off with a single stripe, and when it is full to add a second stripe, and so on. In the case of a relation whose attribute values could be stored in a single stripe, inserting a new tuple an arbitrary position was simple; every other tuple of the relation could be shifted over to accommodate it in parallel. In the realm of external relations, however, inserting a tuple into the first set of stripes of a relation might require the system to read in every other stripe from secondary storage simply to shift a single tuple from the first set of stripes to the second, and a single tuple from the second to the third, and so on.

One alternative, which lies at the heart of this thesis, is to use partitioning. Each bucket could be defined to contain all the tuples with a given range of values for the key attribute. These ranges would be non-overlapping. When the stripes of the bucket were full, then the bucket is split into two buckets. Some primary key value which is contained in the original bucket’s range is selected, and all the tuples which are less than the split point are stored in one of the buckets, and all of the tuples which are greater than the split point are stored in the other bucket.

Strategies for partitioning will be discussed in the later chapters. Here we will
examine the simpler case, where no partitioning is used. The constraint which started off this discussion — that tuples be stored in order of their primary key field — is further weakened:

- *All the tuples which are contained in the same bucket will be stored in order of the primary key, unless the user specifically requests that they be sorted on some other attribute. No guarantees are made about tuples which are contained in other buckets, either in terms of duplicate tuples or duplicate primary keys.*

5.3 External Relational Operators

When examining the performance of our relational operators, we will be primarily concerned with the I/O load on the system; that is, the number of stripes that must be read in and out of secondary storage. We will assume that the Connection Machine's main memory is large enough to hold two input buckets and an output bucket; beyond that, no guarantees are made. Specifically, these algorithms make no assumptions about how data is organized between buckets.

5.3.1 Selection and Deletion

Since we are refraining for the time being from partitioning, both selecting a set of tuples that satisfy a given query and deleting them will require the system to sequentially examine every bucket in the relation. When examining a bucket, we must serially load every stripe which corresponds to an attribute specified in the query predicate. Once all the attributes in the query have been examined, if a single tuple in the bucket satisfies that query all the stripes of that bucket must be loaded in to retrieve the remaining attribute values of that tuple. The expected I/O load on a relation with \( b \) buckets and \( k \) attributes, is \( T(n) = \Theta(b \ast k) \) i/o operations, where
$b \times k$ is equal to the total number of stripes used to contain the relation.

Because the number of buckets will be the primary factor in determining the cost of the relational operators, it is beneficial to store the relations in as few buckets as possible. When deleting tuples, the system should compact the tuples to remove empty spaces in the buckets. All this requires is that when deleting tuples from a bucket, the previous bucket is kept around in main memory. Tuples which have escaped deletion are used to fill the gaps in the previous bucket. If the previous bucket is able to take all the tuples from the current bucket, then the current bucket is discarded.

In order to ensure that we do not inadvertently store tuples with the same primary key in the same bucket, we do not blindly stick the new tuples into any available spaces. We use the merge procedure from above to combine the two sets of tuples and remove any duplicated primary attributes. Since (as specified above) we are primarily concerned with I/O cost, this will not be a major factor. The final cost of deletions will be $T(n) = \Theta(b \times k)\text{i/o} + \Theta(b \times \log(p))\text{cpu cycles}$.

5.3.2 Other Relational Operators

Projection, union, intersection, difference, nested-loops join, and sort-merge join are all executed with algorithms similar to the ones described in the preceding chapter. The distinction is that here they are forced to rely on the external sorting algorithm described above. The difference is in cost; when the number of tuples in a relation grows substantially more than the number of processors, we see that the time required for us to execute relational operators approaches $\Theta(n \log(n))$, the same as that for sequential serial algorithms.
5.3.3 Join

There are some methods for efficiently performing an external join operation that are not feasible in the case where a relation is small enough to fit in an associative architecture’s main memory. Both the hashing join and the semijoin are similar in that they use hashing as a pre-processor to reduce the amount of I/O needed. Where the two differ is in the target: whereas semijoins are used to filter out tuples from the execution of the join that have no compatible tuples in the opposite relation, hashing join seeks to divide the operation into a number of smaller joins that can presumably be performed independently and in main memory.

Hashing Join The hashing join algorithm yields the best expected time performance for sequential systems, as it can theoretically process an equijoin in linear time. This works by reducing a large join operation to the sum of smaller, independent join operations. Tuples from the source and target relations are both hashed on the join attribute into a set of buckets. This hashing yields two sets of sub-relations $R_i$ and $S_j$. Any pair of tuples from R and S which join on this attribute belong to sub-relations with the same index. Any tuple from the target relation that the source relation might want to join with would by definition be located in the same bucket. This eliminates the need for carrying out numerous I/O operations as the tuples from different buckets have to be compared. Within a single bucket, a simple nested-loops join could be used.

This is the method of choice for MIMD architectures, however, it is difficult to implement on a SIMD architecture. One major obstacle to the algorithm is the selection of an appropriate hashing algorithm. If the tuples are not evenly distributed, then it would be difficult to guarantee that buckets will not be filled with more than their capacity of tuples. Although this can be overcome with the allocation of a special “overflow” partition, there is a more fundamental problem at hand. If we make the natural correspondence between stripe and bucket, every time we want to
hash a tuple to a particular bucket, we have to use an I/O operation. To hash the set of tuples in a single stripe could require as many as $p$ I/O operations. If we use some merge algorithm to "percolate" tuples to the appropriate buckets, then we're doing the same work as sorting.

**Semijoins** Hashing can be used in other ways to reduce the necessity for I/O. The *semijoin* technique uses a hash table to filter out all the tuples in both relations that will not be "involved" in the join because there are no tuples in the other relation with the same join attribute value. This suggests a natural implementation on the Connection Machine; two stripes are allocated, one to represent the source relation, and one to represent the target. The two stripes will each contain a single bit in processor $i$ indicating whether any of the tuples in the corresponding relations have a join attribute value which hashes into bucket $i$ of some imaginary hash table.

**Pre-processing Phase** The source relation's stripes which contain the join attributes for all the tuples are sequentially input. Each processor simultaneously hashes its join attribute value, determines which bucket the value would go into, and sets the bit of the processor which corresponds to that bucket to $t$. Then the target relation's join attribute stripes are read in, and the process is repeated for each stripe.

**Processing Phase** The system now uses one of the other join algorithms to carry out the join. When a partition is read in, the attribute values are again hashed. If a processor's tuple hashes to a bucket for which the corresponding stripe representing the other relation is empty, that tuple is removed from the relation.
Chapter 6

Partitioning

One of the chief obstacles that must be overcome in the design of a back-end relational query processor is the “I/O Bottleneck.” Whether one is using a conventional, sequential computer, or an associative, parallel architecture, the fact remains that for all real databases, the volume of data is orders of magnitude greater than the main memory capacity of any physical platform. When answering a query, no matter how fast the processor is, its performance will be determined primarily by the speed with which it can swap data in and out from secondary storage.

This leaves the database designer with only one real option for obtaining significant performance improvements: since there is little to be gained by faster processors, the only alternative is to examine ways to structure the data so as to reduce the I/O load on the system.

6.1 Overview

The problems caused by the “I/O Bottleneck” have been known for a long time. Designers of conventional database management systems (eg. those systems implemented on uniprocessor systems, such as INGRES, DB2, etc) for the most part rely upon indexing as a data management scheme [Dat86]. Using B-trees had several
drawbacks, not the least of which was the fact that B-tree systems could not efficiently be indexed on more than one attribute, and it was not always apparent which attribute of a relation the system should index a relation on. A multi-dimensional or multi-attribute partitioning method would allow a system to partially order tuples along several attributes simultaneously.

Even massively parallel architectures such as the Connection Machine, with their ability to examine tens of thousands of records at a time, are not immune from this disease. Stone, in his design for a text search and retrieval system on the Connection Machine [Sto87], noted that not only were indexes useful in reducing the amount of work necessary to answer a query, but that in some cases, a single processor using a partitioning method could find an answer to a question faster than the 64K processors that are running in parallel.

These results support the conclusions made in the aftermath of DIRECT, when DeWitt studies the I/O Bottleneck in that system. Although Stone’s performance figures were obtained on different types of data than is utilized in relational databases, the message is clear: when designing a database system for parallel architectures, the architect should not forego the use of indices. Designers of algorithms for parallel architectures should not solely be concerned with extracting amount of concurrency that is theoretically possible; it is equally important to use indexing to limit I/O while retaining some degree of parallelism.

It is important to note that the smaller the result size, the larger the benefit of partitioning. The dramatic gains we can obtain from an index for short queries are diminished for large queries because, by definition, a large proportion of the records in a relation have to be examined regardless of the partitioning.
6.1.1 Goals of Partitioning

There are several ways in which a data partitioning scheme can aid us in reducing the cost of relational operations. These benefits of structuring our data fall into two broad categories.

**Pre-processing as filtering** The first, and most important, goal is to reduce the need for costly data transfer in and out of our database computer. Intelligent partitioning schemes can reduce the I/O load, by eliminating as many tuples as possible from consideration for the answer of a given query without having to incur the expense of loading them into the main memory.

**Reducing intra-tuple comparisons** There is a second way in which partitioning can help us, and that is by reducing the amount of computation our database system must perform on the tuples that have been selected. If our relational operators are presented with a query that can be answered by decomposing it into a number of sub-tasks that can be executed independently, presumably the cost of executing that query will be reduced. If we keep tuples that are likely to be compared in the answering of a query together, then we may be able to reduce the number of buckets that must be present in main memory at they same time, and thereby reduce the flow of data in and out of secondary storage.

Minimizing the need for comparison between tuples in different buckets, then, reduces the demands on the I/O system as well as reducing the demands on the query processor. For example, when sorting a large relation which is comprised of many buckets, it would be convenient if we could decompose the sort into a set of independent operations. It would be ideal, for example, if we could simply sort the buckets independently and be assured that the relation as a whole was sorted.
6.1.2 A Sample Partitioning

The approach taken here is to form partitions of the data space, where each partition corresponds to a small range of possible values for each attribute. In figure 6-1, we show a relation partitioned on two attributes. The tuples are represented by points in the data space, and the partition boundaries by horizontal and vertical lines. A one-to-one correspondence will exist between partitions and buckets; one bucket will contain all the tuples whose attribute values fall within a given partition of the data space.

When performing a selection query, we observe the first benefit of partitioning: a reduction in the number of tuples that must be examined. In figure 6-2, we show a representation of a directory for the relation in figure 6-1. The bounds of all of the partitions are available, although the particular tuples contained in those partitions are hidden. A selection query might specify some region of the space. The task of locating a particular record (*get me the record containing the directions customer Fassler gave us for Account #19*) is accomplished by first using the directory to locate the particular partition the record stored in, loading that partition into the Connection Machine, and then performing an associative query match in parallel on all the tuples in that partition.

When performing projections, we see the second goal: although we do not reduce the number of tuples that must be examined, we do cut down on the number of comparisons that must be made between tuples in different partitions. If we were to project the relation shown in figure 6-2 along the X axis, for example, tuples contained in partition A could not be duplicates of tuples in partition I.

In the previous figure a number of assumptions were made that should become explicit. They concern the nature of how tuples are assigned to buckets and how the buckets are split:
Figure 6-1: A relation partitioned on two attributes.

Figure 6-2: An exact match query, retrieving partition F.
Order-Preserving Vital  A key assumption we will make is that a vital characteristic of any storage structure used for multi-attribute access is that the structure should be order-preserving. That is, that records which have similar values for any attribute should be stored near each other, since they are likely to be accesses together. For example, a good data structure for storing our customer-accounts should keep the records in roughly alphabetical order on the customer’s last name to facilitate locating a given customer’s records. But records which correspond to people who share the same accounts should also be kept in close proximity.

Split and Merge Functions Refinements are triggered by bucket overflows if all the records in that bucket are kept in a single block. This is a relatively rare occurrence. The simplest policy is to choose the dimension according to a fixed schedule (eg. cyclic), although you might want to favor certain attributes, which will increase the precision of answers to queries on favored attributes. The location of the split point is at the midpoint of an interval, but could be otherwise, like at the median. As for merging, an appropriate hysteresis factor should come into consideration to avoid the situation where a pair of partitions are repeatedly merged and split. Another issue for merging is the question of how to decide which partitions can be merged together.

Many papers have been written proposing many different forms of storage structures. But although it may seem a trifle obvious that the design of these data structures must be compatible with relational database operators, many systems failed to address the issue of how to incorporate them into relational database management systems. Specifically, little analysis has been done into algorithms for exploiting the partitioning of the data into more efficient relational operators.

The issues we must address are threefold:

1. How should the data be partitioned?
How should we represent that partitioning for optimal access to the tuples contained in those partitions?

5. How can we use the partitioning in our relational operators?

6.2 Multi-dimensional Storage Structures

Many proposals have been put forth for multi-dimensional storage structures for use in database systems. All the different schemes are essentially similar in flavor, but there are some significant differences.

- Partitions cannot grow indefinitely; that defeats the purpose of having them. When inserting tuples into a partition, at some point the partition will become full, and it will be necessary to split it into two (or more) partitions. All known partitioning methods make the split along some hyperplane; that is, they divide the set of tuples into those that are greater than a given value for a given attribute and those that are less than a given value for a given attribute. There are two ways to go about making the split. A local approach makes the division at the median value for some attribute of all the tuples in that particular partition. An alternative mechanism is to use a hash function to make the split.

The local approach organizes the partitions around the specific distribution of the data, similar to the way trees are formed. Boundaries between different regions in the search space are added depending on the values of the data that must be stored. The second method draws the region boundaries for all partitions at fixed places regardless of the distribution of records. These files adapt to the variable content of a file by activating and deactivating boundaries.
Some sort of mechanism is needed to locate the partitions that are involved in a query. In conventional database storage structures, multi-way trees are used to narrow down a search. Many of the multi-attribute hashing systems, on the other hand, go to great lengths to avoid the expense of traveling down the nodes of a tree.

What effect does the splitting of a partition have on either the structure of other partitions or the time required to access them? If only full partitions are split, then the splitting of a particular partition should have no effect on tuples contained in other regions of the database.

How does the method determine when to split a partition? If it does not automatically trigger a split upon overflow, how does it handle that overflow? And then how does it determine which partitions to split when it does decide to do so?

The first distinction is perhaps the most important, which divides the data structures into two broad classes, as defined in [NHS84]: those techniques which organize the sets of data and those which organize the data space those records are chosen from.

### 6.3 Multi-dimensional Trees

K-d trees [Ben75, Ben79, BF79], k-d-B trees [Rob81], and predicate trees [VV84] are all examples of multi-attribute storage structures that fall into the first category. Like B-trees and binary trees, their structure is based upon the principle of divide-and-conquer; each node in the tree divides its sub-nodes into two (or some small fixed number of) regions, where all the values for a particular key on one side of the partition boundary are less than all the values of the particular key on the other side.
The locations of the partitions are determined by the content of the data to be stored. This determination is a local one, as the location of a split point along one branch of the tree does not affect the location of the split point along another branch. The goal of all of these trees is to make the size of the sub-regions as equal as possible.

**K-d Trees** All the nodes on a given level of a k-d tree partition their sub-regions along the same attribute. The attribute for a given level is chosen cyclically. Although selection can be done fairly efficiently with k-d trees, maintaining a dynamic structure is expensive, as deletions cause problems. Also expensive is balancing the tree; unbalanced binary trees, no good algorithms exist for balancing k-d trees. Predicate trees with the clustering method used in SABRE [Che86] are very similar to k-d trees, but are more flexible. Since both methods yield partitions which are not symmetric with respect to all attributes, their performance is sensitive to the ordering in which relations are partitioned.

**K-d-b Trees** Robinson's k-d-b trees are multi-dimensional generalizations of B-trees. The leaf nodes of trees are pointer pages that contain pointers to records which correspond to regions in k-dimensional space. Internal nodes are region pages which reflect the partitioning of a region into non-overlapping, jointly exhaustive sub-regions. The tree root represents the initial partitioning of the entire k-dimensional space. k-d-b trees are always totally balanced, in the sense that the number of nodes accessed on a path from the root node to the leaf node is the same for all leaf nodes.

### 6.4 Multi-attribute Hashing

Multipaging, dynamic multipaging, grid files, multi-dimensional order-preserving linear hashing (MOLDH), and dynamic order-preserving partitioning (DYOP) are all examples of address computation techniques that draw their region boundaries at
fixed places regardless of the contents of the file.

**Multipaging**  Multipaging [MO82], one of the first order-preserving multi-attribute hashing schemes proposed, is also the simplest. Multipaging divides the range of values of each attribute of a database into *intervals* such that approximately the same number of records have values of that attribute in each interval. The intersections of the intervals of all the attributes imposes a grid of hyper-rectangles in space. When some partition becomes full, the interval which corresponds to the projection of that partition along some attribute (usually chosen cyclically) is split.

Segment boundaries are all physical, and cross the entire search space. To partially avoid the arbitrary emptiness that this implies, he proposes that in dynamic multipaging, before splitting, the system checks to see whether it can avoid splitting by shifting a boundary. If not, then the system chooses the axis with either the smallest range of values per partition, the axis that will affect the load factor the least, or whatever and split on it.

The advantages of multipaging are that it is order-preserving, and no directories are required to locate a particular partition. The biggest problem is that the splitting of the interval triggers splits, not only in the overflowing partition, but throughout the data space. The number of partitions increases exponentially, since even un-overflowing partitions are split, resulting in very un-even distributions of records into partitions.

Dynamic multipaging is an extension of multipaging which is intended to reduce the degradation caused by insertions and deletions. Partition boundaries are mobile. When an insertion attempts to place a record in a full partition, the system can attempt to move the boundary and smooth out the distribution of records into partitions rather than unnecessarily executing an expensive split.

Dynamic multipaging has all the benefits of multipaging, and also results in fewer
Figure 6-3: A file partitioned into a k-d tree.

Figure 6-4: A file partitioned with Dynamic Multipaging.
partitions of un-overflowing regions. However, no good algorithms exist for determining \textit{how} to move the boundaries around to limit the number of splits, and clearly the cost of moving boundaries around is non-trivial. And in any case the fact that un-full partitions are still split, albeit less frequently, results in not only an expensive split procedure but also a poor load distribution.

\textbf{Grid Files} The grid file [NHS84] closely resembles multipaging in that it is multi-dimensional and order-preserving. A grid-shaped directory is also used to keep track of partition boundaries. The difference is that several "virtual partitions" of the directory may be stored in the same physical page of secondary storage. Small linear scales, or "axial directories," are used to locate the proper grid boxes. Pointers are followed from the grid boxes to the proper data pages.

Although this method reduces the time required to access records, the space requirements are greater. More significantly, the cost of splitting a partition begins to resemble that of multipaging, since a division of a single partition results in the formation of an ever-increasing number of virtual pages.

\textbf{Multi-dimensional Order-Preserving Linear Hashing} Ouskel [Ous83] describes a hashing method which is an extension of Litwin's linear hashing method [Lit80, Lit81]. It has several properties of use to us: it is order-preserving, like multipaging, but only a single partition is split when overflow occurs. Also, partitions can be accessed directly, without the need of any large directory or index. Several concessions are needed to achieve this, however. Like linear hashing, which also does not require the use of an index, partitions are split in cyclic order, whether or not they are the actual partitions which overflowed. Therefore he had to introduce overflow buckets, and deal with the decreased search time performance which resulted.
Dynamic Order Preserving Partitioning  Dynamic order-preserving partitioning [Ozk85], which was based upon Ouskel's MOLDH, eliminates the need for overflow buckets and chaining which hampered its performance. Partitions have numerical identifiers, which are uniquely determined by their position in the data space and the level at which they were created. Like grid files, many virtual partitions in the directory (here called "implicit" ones) can be embedded in the same physical partition.

A complicated retrieval algorithm is needed to navigate through DYOP's multi-level directory structure. Rather than simply using pointers, as in the grid file, partitions have addresses. Locating the physical partition that corresponds to a virtual partition is no simple matter. In order to locate a partition with a given address, it is necessary to locate the parent partition in the next highest level, until the top of the directory is reached. Then, when at the top of the directory, the physical partition corresponding to the address of the next-level partition is determined, and so on back down to the bottom.

Much has been written about the different methodologies; in the following chapter, these methods will be analysed for their suitability as starting points for relational databases.
Figure 6-5: A Grid file directory. The darker partitions represent physical page boundaries.

Figure 6-6: The same set of points as in the multipaging example, but here, no partitions are split unless they are overflowing.
Chapter 7

RMAP Partitioning and Relational Algorithms

Several goals will be accomplished in this chapter:

- The hash-partitioning/tree-partitioning controversy that was discussed in the previous chapter will be resolved, as the hash-based methods will be shown to demonstrate significantly poorer performance when the data is not evenly distributed.

- We will define the RMAP Partitioning structure, which will consist of a set of partitions together with a directory for efficiently accessing those partitions. A simple example of a relation that is partitioned using this data structure will be presented.

- Algorithms for implementing relational operators for a database partitioned using the RMAP Partitioning method will be described. These methods will be shown to be no worse than, and in most cases significantly better than, those algorithms for performing the same algorithms without partitioning. Also presented will be a method for converting a large collection of tuples into a RMAP structure in one swoop instead of inserting them one-at-a-time.
7.1 Tree-based vs. Hash-based Partitioning

The design of a partitioning method will depend greatly on the assumptions we make about the nature of our data. One of our primary goals of partitioning has to be to maintain as high as possible the occupancy of the partitions. The fewer the number of partitions, the faster our algorithms — such as selection, insertion, and deletion — will go, because it takes the same time to load a half-full stripe into the Connection Machine as it does to load a densely packed one.

Using the benchmark generator (described in chapter 7), a simulation was run to observe the file growth and load factor that resulted from inserting large numbers of tuples into both a hash-style partitioning directory and a tree-based method. Figure 7-1 graphs the comparative results of creating a database using the two methods. In both cases the simulation was performed on a Symbolics Lisp Machine running the Starlisp Simulator, software which emulates the Connection Machine. The benchmark was performed with 2048 processors (the maximum allowable number) and 200,000 tuples were inserted — yielding an approximate 100-1 tuple-to-processor ratio. As we can clearly see, there is virtually no performance difference in terms of load factor between the two methods.

Note the amount of time that the two partitioned relations spend in the balanced state where they have exactly $2^n$ partitions for some $n$. in figure 7-1. However, shortly after one of those periods of quiescence comes a period of rapid growth. Because the tuples are evenly distributed, it seems intuitive that two partitions with the same volume should overflow at the same time. Carrying this argument forward, if there were two partitions $x$ and $y$ such that $x$ was on a higher level of the tree than $y$. In the general case, this would mean $x$ covered a larger surface area than $y$. As a result, incoming tuples would be biased towards $x$ as opposed to $y$. Once too many tuples were attracted to $x$, $x$ would have to split himself until he and $y$ were the same size.

Ouskel [Ous83] offered experimental evidence that given an even distribution of
tuple values across all the relevant dimensions, a partitioned relation \( R \) could be reasonably expected to achieve an occupancy of approximately 65 percent using multi-attribute hashing. However, in the worst case DYOP partitions can have arbitrarily low occupancy, and could end up with only one or no tuples inside. This is no major issue for conventional systems with small page sizes, but when we commit up to the equivalent of 64,000 * (the size of a single tuple) bytes to a single partition, they had better not be empty.

A relation with some measure of correlation between its attributes is shown in figure 7-2. It might, for example, represent the relationship between employee age and salary. The relation contains 24 tuples. Again, we are assuming our database machine has only four processors. In the top half of the figure the tuples are partitioned with a tree-based method, a structuring which only requires eight buckets, for a load factor of .75. Below that the relation is partitioned using some hash-based method such as DYOP. Note that 14 buckets are needed to hold the two dozen records, yielding a load factor of just .375 — half of the tree-based case.

To test this hypothesis, the benchmark generator was run again. This time, normally distributed data was used, with varying means and standard deviations. In the case where the mean on a given attribute was not located at the midpoint of the data space, this resulted in tuples which were densely clustered in one quadrant and sparse in another. Again we used the simulator with 2048 processors and 250,000 tuples. As was postulated above, hash-based partitioning methods are poorly equipped to account for un-even distributions of data. In figure 7-3 we see tuples that contain two attributes, each of whose domains have a median of 75,000 and a standard deviation of 10,000. The range of the data space (the maximum allowable tuple value) is only 100,000. The hash-based method has over 25 percent more buckets to contain the same number of tuples as in the tree-based case. Repeated trials bore these figures out.

In the event of a poor tuple distribution that is too much for even a tree-based
method to bear, it is conceptually simple (although admittedly not cheap) to re-
balance a k-d tree (this can be accomplished in roughly $\Theta(d/p \times \log(d/p))$ steps to
re-balance a sub-section of size $d$). For multi-dimensional hashing, no good methods
exist for "balancing" an entire database, let alone a sub-section of it. In fact, no good
description exists of what a "balanced" multidimensional hash would look like.

7.2 Directories and Balancing

Much of the discussion about the various multi-dimensional storage structures de-
scribed above focuses on the different methods they propose for limiting the growth
of their directory structures. We see this in several places:

Balancing in K-D Trees Bentley refers to the lack of an efficient method for dy-
namically balancing k-d trees as the most outstanding issue that needs to be
dealt with.

Boundary Lines in Grid Files One stated advantage of grid files is that one bound-
ary line does the work of many boundary lines in the k-d tree. The expected
gain is a smaller directory that will be easier to search.

Partition Numbering in DYOP The only distinction between grid files and DYOP
partitions is that the grid file explicitly represents partitions (both implicit and
explicit ones) with a tree-like structure. DYOP, on the other hand, uses a sys-
tematic partition numbering method in conjunction with a smaller associated
tree.

The fact is, we can avoid the use of tree-like directories entirely with the Connection
Machine. By utilizing its associative hardware, we can simultaneously represent
the boundaries of tens of thousands of partitions, more than almost any conceivable
database could ever dream of having to use. And we can access the coordinates of any one or more of those partitions in a single step.

In effect, what we have is a 64K-way tree as the root node of our partition directory. But rather than taking this viewpoint, we will ignore the directory for now except to state that it can do everything we ask of it in a single step, whether it is growing, shrinking, or selecting. Instead of focusing on the growth of a multi-level directory, we will concentrate solely on the partitions themselves.

### 7.3 The RMAP Directory Structure

Now that we have examined the desirable characteristics of a multi-attribute partitioning method, it is time to demonstrate one and display algorithms for performing relational database operators on it. To summarize the results of the previous chapters, we will forego the use of tree-structured directories (for the time being), and will rely on the associative hardware to instantly calculate and return the addresses of the desired stripes. This methodology will be entitled *Regional Multi-Attribute Partitioning* to emphasize that we are focusing on the regions formed by the partitioning as opposed to the structure of the tree directory for accessing those regions.

An RMAP Partitioning is composed of two parts: a list of buckets and a directory which defines the mapping of partitions to regions of the search space. In the frontend, the system maintains a mapping of partition IDs to buckets by storing pointers to those buckets in a list and using the position in that list as the ID. In the Connection Machine’s main memory we associate with each partition a processor that knows the ID of that partition and the bounds of the region it represents. Locating the partition that corresponds to a particular point in space, then, can be accomplished in constant time. This directory will take up \( k \times \log(b) \) bits per processor, where \( b \) is the total number of buckets in the relation. There are two classes of operations that the
system must support for interacting with the directory. Update operators are those which are used to change the directory in the face of insertions and deletions into the corresponding buckets. Selection operators serve two purposes: they are used to locate particular partitions given a complete (or partial) specification of the range, and they are used to "pair-up" associated partitions for ordering sorts and other relational operators.

**Split-Partition** Given a partition ID, an attribute, and a value, split the region corresponding to that partition in two along that attribute at value. Label the region which is less than the split value with ID, and label the new region with new-ID. If the value is not in the range for the partition, do nothing.

**Merge-Partition** Modify the region corresponding to partition ID to be the union of its range and the range of partition NEW-ID. Remove partition NEW-ID from the directory. If the two regions are not adjacent, do nothing.

**Find-Partitions** Given some specification of a region in the data space, return the IDs of all that partitions that overlap the region.

**Split-Partner** Given a partition ID, return the NEW-ID of the partition which was either (1) the most recent partition split off from this one, or (2) if none exists, then the partition which split to form partition ID.

**Partitions-Overlap** Given a pair of partition identifiers, tell whether or not the projection of their regions along a given attribute overlap.

### 7.4 Partition Dynamics

Figure 7-5 illustrates the dynamics of the RMAP partitioning structure on a four-processor machine. In the beginning, all the tuples fit into a single partition. This corresponds to the internal case previously discussed. However, insertions into the
database will overflow the partitions which have a capacity of only four tuples. The relation only has two fields that will be of interest to the system as potential partition attributes.

1. In the first panel of figure 7-5, we demonstrate the state of the system when partition 0 filled to capacity with four tuples.

2. When inserting a fifth record into the database, partition 0 will overflow. The partition must be split along some dimension. For now, let us assume that the choice of dimensions is cyclic, starting with the x-axis, and that splits are made at the midpoints of the data contained in the partition (a further discussion of split policies will be held later). To calculate this median, we sort the records according to the value of their x attribute using some appropriate internal sorting method and take the value contained in processor p/2. Records whose attribute value for x are less than the median remain in bucket 0, while the remaining records are inserted into bucket 1. This splitting can be accomplished in a single step, the broadcast of the median to all the processors.

3. We pack the records into the low-ordered processors. This will aid us in further insertions. Note that the tuples are all sorted on x.

4. We continue to insert records, and when inserting tuple z, bucket A overflows again. We split 0 again along the y-axis, creating bucket 2. This split has no effect on 1.

7.4.1 Deletions

When tuples are deleted from our database, it might be necessary to re-combine partitions that have been split in order to prevent the bucket occupancy rate from dwindling down arbitrarily. When a set of records are removed from bucket $B_1$, the
system queries the RMAP directory to determine the identity of the partition that was the split partner ($B_2$) of the slimmed-down one. If one exists, and the sum of the capacities of the two partitions is less than the number of available processors, then the two buckets can be merged into a single one. If the split partner of $B_2$ is equal to $B_1$, then $B_1$ and $B_2$ are essentially two siblings – leaves on a binary tree that share the same parent. In that case, $B_1$ and $B_2$ can be merged together and the tuples in the corresponding buckets combined into one. In figure 7-6, we see a sample partitioning where some partitions have compatible siblings and others' siblings have already been split further.

If $B_2$'s split partner is not $B_1$, then the two partitions cannot be merged. This is comparable to the case where a leaf in a binary tree has a sibling that is not a leaf. The total capacity of all the leaves descended from that sibling node must be greater than a single bucket, or else they would have already been merged together.

More complicated schemes would be necessary for balancing a partitioned file in the case that a bucket without a compatible sibling was sparsely occupied. In figure 7-6, if $A$ were to be gradually emptied out, no other partition is available as a merge candidate. Re-balancing at some higher node of the tree, using the methods detailed in section 7.4.2 for large-scale dynamic activity, could be used in this case.

### 7.4.2 Large-Scale Dynamic Action

In addition to a method for handling sequential insertions, we need a means of efficiently partitioning a large block of records. This would be needed to convert an existing database to one that is partitioned on multiple attributes. Such a method would be vital for testing purposes, when the cost of inserting hundreds of thousands of records into a database one by one would be prohibitive. The key function we need to accomplish this is a median-finding algorithm.
**Linear-Time Medians** In order to partition our database, we need a method for efficiently determining the median of a set of integers. Here we present a parallel algorithm called *find-nth*, which is a concurrent variant of the *pick* method described by Blum in [Blu73]. Let \( s \) be the number of stripes, \( n \) be the number of tuples, and \( p \) be the number of processors. Let \( i \) be the index of the element we are looking for.

*Find-nth* works by partitioning the database into three parts: those pieces of data that are known to be less than \( i \), those pieces of data that are known to be greater than \( i \), and the remainder whose positions cannot be determined. The function works similar to quicksort; data is partitioned around some value, and the unneeded portion is discarded. To ensure the linear-time performance of the algorithm, we must choose a partition element that is known to be somewhere in the middle of the file.

This algorithm is well-suited to implementation on the Connection Machine, because it relies on two operations: the internal sorting of a partition, and the distribution of a single result for comparison with many others.

1. Find the median of each of the \( s \) partitions. Collect the sub-medians; it is fairly safe to assume that we can collect them into a single stripe (as long as there are fewer than 16,000 partitions).

2. Compute the median of the \( s \) sub-medians. Call this \( m \). We now have the situation shown in figure 7-7: at least \( n/4 \) elements are less than or equal to \( m \), and at least \( n/4 \) elements are greater than or equal to \( m \). Call the stripes whose medians are less than \( m \) the set \( l \), and those stripes whose medians are greater the set \( g \).

3. If \( n/4 \) is greater than \( i \), then for each stripe in \( g \), we discard all the elements whose rank is less than the median's. Alternatively, if \( n/4 \) is less than \( i \), we discard from each stripe in \( l \) every element whose rank is less than the median's.

4. Calculate the number of items discarded. This could be accomplished in a single
step, accumulating the number of tuples discarded from each processor. Call this \( d \).

5. Recursively call \( \text{find-nth} \) on the remaining tuples, with the new \( i \) equal to \( i - d \) if elements from \( I \) were discarded, and \( i \) if elements from \( g \) were discarded.

**Performance** The first step will require \( s \) I/O operations, and \( s \cdot \log^2(p) \) computation cycles. The second step requires no I/O and \( \log^2(p) \) computation cycles. The third step takes an additional \( s \) I/O operations and \( \log(p) \) CPU cycles to compute the summation. This gives us a recurrence of:

\[
T(n) \leq T([3s/4]) + 2 \cdot s \cdot io + 2 \cdot \log^2(p) \cdot cpu
\]

\[
T(n) = \Theta(s \cdot (io + \log^2(p) \cdot cpu))
\]

\[
T(n) = \Theta\left(\frac{n}{p} \cdot (io + \log^2(p) \cdot cpu)\right)
\]

### 7.5 Relational Algorithms

As stated previously, there are two classes of operations that our RMAP directory must support: associative retrieval and relational database operations. We differentiate between those that operate on a single relation — insertion, simple deletion, sorting, simple selection, projection — and those which operate on multiple relations — join, union, intersection, deletion, complex deletion, and complex selection.

#### 7.5.1 Single-relation Algorithms

**Phase I: Filtering** Eliminates all partitions whose tuples cannot be needed to answer the query.
**Phase II: Matching** Determines which partitions must be compared at some point. When performing a projection, two partitions with non-overlapping regions on the projected attribute do not have to be compared with each other, although both partitions must be examined separately.

**Phase III: Query Execution** Involves applying the internal relational operators to partitions and using merge functions to combine the results into longer and longer runs of processed partitions.

**Phase IV: Cleanup** Formulating the results of our queries into full-scale relations. Since most relational operators output their results in some sort of sort order, this should not be too difficult. In the worst case, this will simply involve re-partitioning of the resultant relations.

The simplest relational queries to answer with our partitioning method are selection queries. The RMAP Directory makes locating a desired partition an single-step operation. Retrieving a set of partitions that match a more complex query can be accomplished almost as fast; the only limiting factor is the number of attributes that must be examined.

**Exact Match** Exact match queries can be answered in a constant amount of time using any form of multi-attribute partitioning, assuming that the relation is large enough that several partitions have been formed. Since each partition has a fully specified range associated with it, and these ranges are non-overlapping, the desired tuple can only be located in one partition. This partition must be retrieved from secondary storage and brought into the Connection Machine’s working memory. Each processor then examines the tuple it holds in parallel, and the results are returned to the host computer. Figure 6-2 showed an example of this query.
Partial Match  To answer a partial-match query, we look at the directory and locate all the partitions which might contain tuples satisfying the query. The number of partitions that might satisfy the query is dependent upon the number of attributes we are partitioning upon, and how many of those attributes are specified in the query.

In figure 7-8, we demonstrate a partial match retrieval, where the user only specified the $x$ attribute value. If we have partitioned on $d$ attributes, and the query only specifies $q$ of those attributes, we can estimate the number of partitions accessed [Ben79] as:

$$P \times \frac{2^{d-q}/2^d}{2^d} = \frac{P}{2^q}$$

Range Queries  Range queries are answered in a fashion similar to that of a partial-match query. In this case, the number of partitions searched is proportional to the size of the query. An example of a range query is demonstrated in figure 7-9.

7.5.2 Sorting

A directory which is able to quickly locate partitions is useful, but more functionality is needed. In order to perform relational operators such as sort and projection, we need a method for pairing off partitions for processing. For this, we resort back to the tree structure that was discarded in the previous chapter. We will not be using this tree for retrieval; it will only be utilized as a method for storing the split history of the data space.

Before sorting a relation, we note that a good portion of the work has already been accomplished. Splits have been made between partitions along the sort attribute, and if our tuple inserter works correctly, two partitions separated by a split along the join attribute should already have their tuples partially ordered with respect to each other.
To sort, then, we begin with the tree in figure 7-10 on level four. Assume we would like to sort the relation on attribute \( x \). Just as with the standard external algorithm, we begin by sorting each partitions’ values individually. Each pair of partitions on the lowest level — \( \{1, 2\}, \{3, 4\}, \ldots, \{15, 16\} \) have been split along the \( y \) attribute. Therefore we have no information about their tuples’ relative values. We must perform a comparison and merge-exchange between the tuples in each pair of partitions, forming runs of length two.

At the next level of the tree, however, we see that the splits have all been made along \( x \). Therefore, we know that for each node of the tree that is an \( x \)-discriminator, the runs of partitions along the left branch contain \( \leq \) values that are strictly less than or equal to those in runs along the right branch. To merge these runs together along the \( x \)-axis, then, requires no work at all. We continue in a similar fashion up the tree, merging partitions at \( y \)-nodes and simply concatenating them together at \( x \)-nodes.

**Projection** Next to selection, projection is the unary operator that benefits most from multi-attribute partitioning. To implement projection using the RMAP directory, we could use the following algorithm which was adapted from the discussion of internal algorithms:

1. Sort the relation upon one of the attributes the tuples of that relation are being projected on.

2. Perform a linear scan on the elements in the result buckets, removing duplicate tuples.

3. Pack the partitions.

But there is a better approach. We can modify the patterns of comparisons made by the sort function to better meet the needs of the project operator. If we are
projecting on a set of attributes \( \{a_1, a_2, \ldots, a_d\} \), and one of those attributes is also an attribute we have been partitioning on, then every node in tree whose discriminator is that node divides the data space into two regions such that any tuple in one cannot be duplicated in the other. These two regions correspond to the left and right branches of the node. This means that there is no need to compare any tuples contained in one of the branches of the tree with any of the tuples contained in the other branch of the tree.

In figure 7.5.2, a three-dimensional data space is displayed. This data space has been divided into seven partitions, labeled \( A \) through \( G \). If we drew out the k-d tree for this space, we would see that the root node is an \( X \)-discriminator, while the next level of the tree partitions on \( Y \), and so on cyclically. Carrying this argument forward, we see that sorting is just a special case of projection when there is only a single projection field.

### 7.5.3 Bi-relational Operators

Our method for answering multi-relational operators is similar to that described for operators which acted on a single relation.

**Phase I: Filtering** Eliminates all partitions whose tuples cannot be needed to answer the query.

**Phase II: Matching** The matching phase involves locating sets of partitions from \( R \) and \( S \) such that each partition in the set from \( R \) must be compared with each partition in the set from \( S \).

**Phase III: Pre-processing** If necessary, the relations are sorted before the relational operator can be carried out.
**Phase IV: Query Execution** Involves applying the internal relational operators to partitions and using merge functions to combine the results into longer and longer runs of processed partitions.

**Phase V: Cleanup** Formulating the results of our queries into full-scale relations.

**Duplicate Detection**

**Union, Intersection, Difference** There are two types of algorithms available for performing these relational operators. The particular choice of an appropriate method will depend upon the characteristics of the data operated on.

1. One of the simplest tasks that can be performed with multi-attribute partitioning is that of duplicate detection. The data structure effectively divides the search space into pairs of partitions that can be compared without having to incur the expense of merging the two relations and sorting them. In figure 7-12, we have two relations that we want to unite and remove duplicates. By superimposing the two trees, we can directly identify those partitions that might have duplicate tuples. In the un-partitioned case, there are 30 possible pairings of a partition from $R$ with a partition from $S$. In the partitioned case, there are only 13.

2. We can use the directory structure to sort and merge the two relations as described in section 7.5.2. After the sort, this method would proceed in a fashion identical to that described in the previous chapter.

**Cartesian Product** Having a directory does not facilitate the execution of a cartesian product, since it by definition involves pairing off all of the tuples one-by-one. There is no alternative but to proceed as in the case where there is no data structure.
Join The biggest winner among the sorting algorithms in terms of use with a partitioned database is the sort-merge join algorithm. Using the method described above as long as the join attribute was one that was partitioned upon, the external sorting phase should proceed much more rapidly. If, however, the join attribute is not one that the relation was partitioned on, the situation reverts back to the no data structures case.

7.6 Observations

Partitioning does not come at the expense of load factor. By partitioning the database using a method which is sensitive to the distribution of the tuples, RMAP Partitioning can achieve load factors as high as 85 percent, even in the face of very bizarre data patterns.

Hash-based partitioning is feasible if the distribution is known. It has also been shown that in the case where the distribution of tuples is even across the data space, the performance of hash-based partitioning methods such as DYOP are indistinguishable from the RMAP method.

Multi-level directories are obsolete. Much of the effort that went into the design of many of these partitioning systems was devoted to devising ways to avoid having to traverse a large tree to locate a partition or a set of partitions. Now that we can have the boundaries of up to 64K partitions in main memory such that the access time to any one of them is the same as to any other, the directory is no longer an issue.

Selection of partitioning attributes matters. The MAP model makes no decisions about which attributes the partitioning will be performed upon; this decision is
left to the user. If a relation is not partitioned on a given attribute, then the directory is of no use in answering queries that specify some property of that attribute. The naive way of approaching the problem of which fields a relation should be partitioned on is to say all of them. Clearly this is a bad strategy, since the performance advantage obtained by sorting with a RMAP directory is proportional to the number of partitions that have been made upon the sort attribute.

Some relational operators can benefit greatly from partitioning. As we have seen above, the performance of a projection can be greatly improved by presenting it with a partitioned database, as long as the attributes the relation is projected on are among the partition attributes. Sorting is improved to a lesser extent, since sorting can be viewed as the degenerate case of a relation projected on but one attribute.

Caveat. One important assumption that is being made here is that the directory tree is relatively balanced. If the tree is severely out of kilter, then sorting using this methodology is difficult. In Figure 7-13, we see a three-dimensional space where the depth of the k-d tree is linear as compared to the number of partitions.

It is not unreasonable to expect that the directory will remain relatively balanced. Since the partitions are made according to the distribution of the tuples, only a significant change in distribution could serve to unbalance the tree. In the worst case, sections of the tree that become unbalanced can be re-partitioned using the methods described above.
Figure 7-1: RMAP vs. DYOP with even distribution of tuples.
Figure 7-2: RMAP is far superior to hash-based methods when there is an uneven distribution of tuples.
Figure 7-3: Load factors observed for insertion with mean of 75,000 and deviation of 10,000.

Figure 7-4: An RMAP directory (and the corresponding tuples) for a relation with 3 partitions and 9 tuples on a four-processor machine.
Figure 7-5: Multi-attribute partitioning in action.
Compatible partitions for merging:  B - C, E - G, F - H, I - J

Partitions without un-split siblings:  A, D

Figure 7-6: Compatible partitions for merging.

at least 25 percent must be greater than M

Figure 7-7: Find-nth in action.
Figure 7-8: A partial match query, retrieving partitions C, F, and H.

Figure 7-9: A range match query, retrieving partitions A, B, and C.
Figure 7-10: A k-d tree indicating where previous partitions have been made.
Figure 7-11: A data space that has been partitioned on three attributes.
Figure 7-12: The union of two relations using multi-attribute partitioning.
Figure 7-13: A three-dimensional partitioning of a relation with uneven distribution.
Chapter 8

Benchmarking

A systematic approach must be taken to the issue of how to quantitatively measure the performance of a relational database management system. Surprisingly few researchers have examined this topic. This is a need which must be addressed.

Most benchmarks of database systems involve the use of data taken from an existing DBMS and the execution of a restricted set of queries upon that data. There is no way to systematically alter the characteristics of the database to provide meaningful execution results. We would like to have a generator that would allow us to specify the nature of our relations and automatically generate a sample database that meets our specifications.

The most commonly thought-of view of benchmarks is that they enable users to compare the performance of two dissimilar systems on a similar set of data. Although this is one use, it is by no means the only one. It is the second use which will most enlighten us on the desirable characteristics of this benchmark. And after examining this conventional view of benchmarks in light of this new purpose, our opinion of what makes a good set of data to compare dissimilar systems should change as well.

This unorthodox view of benchmarks is that they should be used as a means for testing and debugging of systems in their development. When testing code, users certainly desire functions which will produce the typical, expected data that it will
eventually be used for. But more importantly, they need to generate boundary conditions, uneven distributions, and generally aberrant behavior in an attempt to attempt to uproot bugs and to probe the limits of the system's robustness in the face of unexpected inputs. So why shouldn't the same characteristics be desired for a system that could systemically compare two different methods for managing databases?

For example, the assertion was made earlier in this paper that multi-attribute hashing methods will fare poorly when the data is not evenly distributed. This could not be reliably verified if our only source of tuples was some pre-existing database. With a benchmark generator, we can compare the hashing method to k-d trees for a range of possible data from evenly distributed data to normal distributions with a large standard deviation to those with a very small standard deviation centered on the boundaries of an attribute's range.

8.1 Characteristics for a Benchmark Generator

In short, the RMAP benchmark generator should meet the following specifications:

**Controllable results** The data values must be flexible enough to allow systematic benchmarking. If characteristics of the data can affect the system's performance, we must have a means of isolating the important characteristics and determine what effects varying those parameters have on the system.

**Repeatable results** Just as with cold fusion, results are not meaningful if they cannot be duplicated.

**Simplicity of generation** A program which generates these benchmarks must be easy to use and must be very rapid, in order to generate a large number of different databases.
Controlled Randomness There are no guarantees that the data taken from an existing system are random, and with all probability they are not. Patterns, which are to be expected in any real data, can significantly affect performance, even if these patterns are not perceived by the users. People are notoriously poor estimators of randomness of data.

But just as important is the ability of users to test database systems on non-random data. As previously explained, many proposed multi-dimensional storage structures depend upon even distribution of the data for smooth growth. Potential buyers should be able to observe what happens when this condition is not met.

Two types of tuple distribution:

**Even Distribution** The probability that a tuple will have an attribute valued between zero and the maximum is evenly distributed.

**Normal** The values follow a bell-curved normal distribution.

Wide range of queries We would like to be able to tailor the characteristics of our sample databases to the parameters of a particular query or set of queries. For example, if we are interested in the performance of a selection, we might be interested in the number of tuples, the number of attributes, the number of distinct values for the primary key, etc. If we were investigating alternative join algorithms, we might be interested in the effects of varying the relative sizes of the two relations, the size of the resultant join, the range of the join attribute, etc.

8.1.1 Structure

Our goal is to provide the maximum possible amount of flexibility in the structure of the sample data. We will simply ask to user to supply design parameters that
the data should satisfy, then construct relations which come as close as possible to meeting those specifications. This is in contrast to other systems which supply users with relations whose fields have quantifiable characteristics, but require the user to determine how to phrase his query to extract the desired performance from the system.

There are two topics which must be addressed:

- What are the characteristics of the individual records to be used?
- What are the dynamics of the relations?

8.1.2 Benchmark Parameters

When selecting the parameters to be used to build benchmark relations from, it is important to be able to modify those characteristics of a database which might have an impact upon some system's performance. There are many characteristics of relations that may change the performance of some of the algorithms discussed previously, or those of competing algorithms not mentioned. For example, here are two sets of parameters a user might wish to specify, one in preparation for a sorting test and the other before a join test.

Sort Query Parameters

1. Relation Size.

2. Range of Sort Attribute. The range of the sort attribute would affect the number of partitions formed when using multi-attribute hashing. It might also be a factor when measuring the performance of radix sorting methods which are dependent upon the number of bits in the key field.
Equijoin Queries

1. Join Selectivity. The number of tuples in the resultant join.

2. R-Selectivity. The number of tuples in Relation $R$ which are used in forming the actual join, or the number of tuples in $R$ which have the same join attribute as some tuples in $S$.


4. Projection Size. The number of distinct join attribute values for the two relations.

5. Block Size. The sort-merge join algorithm divides up the two relations into sets of tuples with equal join values, and then performs a cross-product on those tuples. The block size is equivalent to the number of tuples produced by one of these sub-cross-products.

Note that we have the following relations holding between the parameters:

\[
\text{join-size} = \text{join-selectivity} \times \text{relation1-size} \times \text{relation2-size}
\]

\[
\text{join-size} = \text{number-of-blocks} \times \text{block-size}
\]

\[
\text{block-size} = \text{relation-1-projection} \times \text{relation-2-projection}
\]

\[
\text{relation-1-selectivity} \times \text{relation-1-size} = \text{relation-1-projection} \times \text{number-of-blocks}
\]

\[
\text{relation-2-selectivity} \times \text{relation-2-size} = \text{relation-2-projection} \times \text{number-of-blocks}
\]

In figure 8-1, we show two relations which could have been produced in response to a request for a join test case. Performing an equijoin on the two relations over the Account Number attribute will result in a query that meets all of the characteristics specified inside the box.
### Relation 1

<table>
<thead>
<tr>
<th>Customer</th>
<th>Acct #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fassler</td>
<td>10</td>
</tr>
<tr>
<td>Garrison</td>
<td>12</td>
</tr>
<tr>
<td>Becker</td>
<td>1</td>
</tr>
<tr>
<td>Zedeck</td>
<td>3</td>
</tr>
<tr>
<td>Inman</td>
<td>19</td>
</tr>
<tr>
<td>Coppola</td>
<td>1</td>
</tr>
<tr>
<td>Labendz</td>
<td>10</td>
</tr>
<tr>
<td>Shandler</td>
<td>12</td>
</tr>
<tr>
<td>Hewitt</td>
<td>19</td>
</tr>
<tr>
<td>Palmucci</td>
<td>3</td>
</tr>
</tbody>
</table>

### Relation 2

<table>
<thead>
<tr>
<th>Acct #</th>
<th>Trans. Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5-13-89</td>
</tr>
<tr>
<td>1</td>
<td>5-13-89</td>
</tr>
<tr>
<td>2</td>
<td>5-13-89</td>
</tr>
<tr>
<td>6</td>
<td>5-13-89</td>
</tr>
<tr>
<td>10</td>
<td>5-12-89</td>
</tr>
<tr>
<td>19</td>
<td>5-12-89</td>
</tr>
<tr>
<td>4</td>
<td>5-12-89</td>
</tr>
<tr>
<td>5</td>
<td>5-12-89</td>
</tr>
<tr>
<td>1</td>
<td>5-12-89</td>
</tr>
<tr>
<td>6</td>
<td>5-09-89</td>
</tr>
<tr>
<td>10</td>
<td>5-09-89</td>
</tr>
<tr>
<td>19</td>
<td>5-09-89</td>
</tr>
<tr>
<td>12</td>
<td>5-09-89</td>
</tr>
<tr>
<td>12</td>
<td>5-09-89</td>
</tr>
<tr>
<td>5</td>
<td>4-1-89</td>
</tr>
<tr>
<td>13</td>
<td>4-1-89</td>
</tr>
<tr>
<td>7</td>
<td>4-1-89</td>
</tr>
<tr>
<td>13</td>
<td>4-1-89</td>
</tr>
<tr>
<td>7</td>
<td>4-1-89</td>
</tr>
<tr>
<td>2</td>
<td>1-1-89</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relation 1 Size = 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation 1 Projection = 5</td>
</tr>
<tr>
<td>Relation 1 Selectivity = .6</td>
</tr>
<tr>
<td>Relation 2 Size = 20</td>
</tr>
<tr>
<td>Relation 2 Projection = 10</td>
</tr>
<tr>
<td>Relation 2 Selectivity = .3</td>
</tr>
<tr>
<td>Join Size = 12</td>
</tr>
<tr>
<td>Join Selectivity = .167</td>
</tr>
<tr>
<td>Block Size = 4</td>
</tr>
<tr>
<td>Number of Blocks = 3</td>
</tr>
</tbody>
</table>

Figure 8-1: Generator-produced relations suitable for equijoin.
8.2 Implementation

The implementation of the RMAP benchmark generator is straightforward, utilizing the Connection Machine to generate random data with the desired characteristics. The tricky part is to scramble the data so as to ensure no un-observed patterns will be present.

In order to begin to satisfy the requirements listed above, it is clear that we need some reproducible method of generating large quantities of random data. Although the data should be random, we need a method of controlling that randomness and quantifying the ranges and types of values contained.

It is quite common for relations to have a key field, which contains items which uniquely identify the records they contain: social security numbers in a personnel file, for example, or a join field. Frequently these relations would be indexed on that unique attribute for rapid access; randomly shuffling these relations, then, corresponds to assigning them all a random unique index attribute.

To accomplish this, we need a means of generating a list of random integers from 1 to \( n \) with the following characteristics:

1. Each integer from 1 to \( n \) is included once and only once.

2. The values are evenly distributed throughout the sequence (the probability that any given integer can be found in any given position is equal to \( 1/n \)).

We accomplish this by generating a stripe of the same length as our relation, all of whose processors contain values between 0 and the number of tuples, such that all of the values are used and none are duplicated.

This need for a method of generating long, random sequences of non-repeating integers is the primary obstacle to the benchmark system. In addition to the obvious
use, random non-repeating strings of integers can also be used for scrambling other
sets of data. This will be used for several things in the benchmark: generating the
unique identifier to be used as the tuples’ key fields, selecting the attribute values
that will be used in the join fields, and scrambling those fields.

8.2.1 Scrambling Method

Assume we had two lists consisting of the elements $1, 2, 3, 4$ and $a, b, c, d$ such that the
order of the elements within each of the lists was randomly determined. We see this
in the first line of figure 8-2. If we had a mechanism for randomly shuffling the two
lists together, preserving the relative orders of the elements from each list, we could
generate a scrambled list of length 8.

Algorithm 8.1 A new algorithm for forming unique random strings of integers. This
algorithm is based on the theorem that if you have two runs of randomly ordered
integers $X$ and $Y$, and if you randomly merge the two of them so that the relative
orderings of the $X_i$ and the $Y_j$ are preserved, it is equivalent to forming a random
string of length $X + Y$.

The method we will use is to sequentially pop an element off one or the other lists
and place the selected element at the tail end of our new list. The probability that a
given list will have its head chosen is equal to the ratio of the number of un-selected
elements in that list compared to the number of un-selected elements in the second
list.

In the next step of figure 8-2, we choose a random number from 0 to the total
number of un-selected elements. Since the random number is greater than the number
of elements in the first list, then processor zero takes the value of the head of the second
list ($d$). In the third line of the diagram, we show a pair of values for processor zero:
$R$, which indicates that the processor will take the head of the right-hand (the second)
list, and a 4, which indicates that the first list still has a length of four when processor zero is finished.

Processor one then generates a random number from 0 to 6, and picks 3. Since 3 is less than the size of the first list (4), processor one takes the value of the head of the first list (3). We continue for all of the available processors.

Why does this work? In figure 8-3, we see the scrambler produce a randomly sorted list of length four. Focusing on one of those elements (x), we will see that the probability that x ends up in any one of the four slots in the final list is equal to \( \frac{1}{4} \).

In step 1, x is merged with y to produce a scrambled list of length two. A random integer less than 2 is selected, and if it is equal to 0 then x is placed at the head of the list. There is an equal probability that x will be the head or the tail of the list produced at the end of this step.

In step 2, we are merging the list (x, y) with another list (u, v) (u and v are not shown). We have two possibilities that we must examine; the case where x was placed before y in the previous step, and the case where y was placed before x.

1. If x was placed before y, then there are only three places that it can go in the final output list (it cannot be the fourth element because that would violate the relative ordering of it with y). These placements are shown by bold arrows, and the probabilities of these placements given that x is the head are also in bold.

To choose the head of the result list, the system selects a random digit from 0 to 3. If it chose 0 or 1, x becomes the head of the new list. The chance of this happening is \( \frac{1}{2} \). The second element of the new list is now selected; there is a \( \frac{2}{3} \) chance that the second element will be the head of the list that the first element was not from (since the probabilities are equal to the ratios of the list lengths). The only way that the second element could be x, then, is if u was chosen as the first element (\( p = \frac{1}{2} \)) and x was chosen next (\( p = \frac{2}{3} \)) for a net
The two scrambled subsequences.

Each processor chooses a random number, with the seed starting at 8 in processor 0 and decreasing to 1 in processor 7.

Each processor holds two values: whether it will receive a final value from the left or the right subsequence, and how many elements remain in the left list when it is through. For a processor to receive an element from the left list, its randomly selected value (above) must be less than the number of remaining elements in the left list.

Figure 8-2: Merging two random strings of length four to form a single random string of length eight.
There is a \( \frac{1}{6} \) chance of \( x \) becoming the third element of the list \( (1 - \frac{1}{2} - \frac{1}{3} = \frac{1}{6}) \).

2. In dashed lines and italics, we see what would happen to \( x \) if it followed \( y \) after step 1. It could not be the head of the output list, since \( y \) must precede it. It could be the second element if \( y \) was the head \( (p = \frac{1}{2}) \) and it was chosen next \( (p = \frac{1}{3}) \) for a net probability of \( \frac{1}{6} \). There are two ways \( x \) could be the third element of the output list: if \( y \) was chosen first \( (p = \frac{1}{2}) \), \( u \) second \( (p = \frac{2}{3}) \), and \( x \) third \( (p = \frac{1}{2}) \), for a net probability of \( \frac{1}{3} \), or if \( u \) was chosen first \( (p = \frac{1}{2}) \), \( y \) second \( (p = \frac{2}{3}) \), and \( x \) third \( (p = \frac{1}{2}) \), for a net probability of \( \frac{1}{3} \). The chances of \( x \) becoming the tail of the list is equal to \( 1 - \frac{1}{6} - \frac{1}{3} = \frac{1}{2} \).

The result of this is that \( x \) has a \( \frac{1}{2} \times \frac{1}{2} = \frac{1}{4} \) chance of being the head of the list (the chance that it is the head of the output list given that it was chosen before \( y \) in step 1 times the chance that it was chosen before \( y \) in step 1. Likewise, \( s \) can be the second element of the output list with probability \( \frac{1}{2} \times \frac{1}{3} + \frac{1}{2} \times \frac{1}{6} = \frac{1}{4} \), the third element with probability \( \frac{1}{2} \times \frac{1}{6} + \frac{1}{2} \times \frac{1}{3} = \frac{1}{4} \), and the final element with probability \( \frac{1}{2} \times \frac{1}{2} = \frac{1}{4} \).

### 8.2.2 Normal Distributions

Given a mean and a standard deviation, Knuth [Knu73] presents a method for returning a set of numbers which are normally distributed with those parameters. A brief description of the "polar method" follows.

**Algorithm 8.2** Generate a set of random numbers which will obey a normal distribution with mean of zero and standard deviation of one.

1. \( U_1 = \text{(random 1.0)}. \)

109
2. $U_2 = \text{(random 1.0)}$.

3. $V_1 = 2 * U_1 - 1$.

4. $V_2 = 2 * U_2 - 1$.

5. $S = V_1^2 * V_2^2$.

6. If $S \geq 1.0$, then go back to step 1.

7. Otherwise, return $V_1 * \sqrt{-\frac{2.0 \cdot \log(S)}{S}}$.

Algorithm 8.3 Generate a set of random numbers which will obey a normal distribution with mean $m$ and standard deviation $\sigma$.

1. Generate $S$, a random number obeying the standard normal distribution.

2. Return $m + \sigma \cdot S$

8.3 Related Work

The closest thing to a “standard” benchmark generator is the Wisconsin Database Benchmark. Perhaps the first attempt to develop a database customized for benchmarking and a set of queries to accompany them, it was developed in 1983 at the University of Wisconsin for the testing of the Britton-Lee Intelligent Database Machine.

Wisconsin Benchmark relations can have either 1000, 2000, 5000, or 10,000 tuples. All relations are comprised of 16 attributes: the first 13 are integers, and the final three are strings. The first two fields contain unique integer identifiers, ranging from one to the size of the relation. Relations will be sorted on the first of those attributes.
Each of fields 3-13, has a different integer specifier (2, 10, 100, 250, 500, 1000, 2500, 5000). The values in these fields range from 1 to the specifier, with duplicates allowed.

Fields 14 and 15 are string equivalent to the first two fields. The elements are 52-character strings, where characters numbered 0, 26, and 51 are chosen from the set A...V. All other characters are 'x', allowing \(2^3 = 10,648\) possible combinations. The strings in either column do not repeat anywhere else in that field; the strings in field 14 are in sort order, whereas they are not in field 15.

Field 16 contains a 52-character string, with the same format as in field 14. The string takes on one of four different values, selected at random: AxxAxxA, HxxHxxH, OxxOxxO, and VxxVxxV.

To use the benchmark generator as a method of obtaining relations with different parameters, users must sit down with a pencil and paper and calculate which fields they have to use to get characteristics closest to what they need.

8.3.1 Problems With the Wisconsin Benchmark

The Wisconsin Benchmark is useful in comparing systems, but its use as a development tool is limited.

**Ease of Use** It is cumbersome to modify parameters to obtain a variety of different query results. The user cannot specify to the generator that a join selectivity of .001 is desired for relations with 10,000 tuples; it is necessary to figure out what combination of pre-packaged attribute fields will yield something close to what is desired.

**Flexibility** A more substantial problem are the constraints that the system imposes upon the user in terms of what parameters can be modified. The generator was de-
signed primarily for comparing different storage structures and indexing schemes. As
such, its (documented) use focused exclusively on the unique identifiers contained in
the first two attributes. When attempting to compare the performance of alternative
join algorithms, we find that the underlying structure of the database attributes lim-
its our ability freely specify the characteristics of our join attributes. Users can only
specify the range of values for a (non-primary-key) attribute, not the distribution of
values within it, since it is assumed a flat, even distribution.

When joining two relations \( R \) and \( S \) on attributes \( R_i \) and \( S_j \), if the range of \( R_i \) is
greater than \( S_j \) then almost every tuple in \( S \) will be involved in the join. This occurs
because the domain of \( S_j \) is a subset of the domain of \( R_i \). Also, if \( R \) is sorted on
\( R_i \), every tuple before a certain point in the file (the maximum value of \( S_j \)) will be
included in the join, whereas every tuple after that point will not be.
Step 1: X merges with Y, taking the first or second position with equal likelihood.

Step 2a: If x came before y when they were merged, the probability of x's placement when that list is merged with another list of length two is shown in bold.

Step 2b: If x came after y when they were merged, the probability of x's placement when that list is merged with another list of length two is shown in italics (follow the dashed lines).

Figure 8-3: Demonstration that the algorithm produces a randomly shuffled run.
Chapter 9

Conclusions

Massively parallel machines like the Connection Machine offer great promise as a relational query accelerator. The associative architecture and the routine network together greatly facilitate searching and sorting, the primary needs of any database system. However, the I/O Bottleneck has always been an obstacle to the effective use of database machines whatever the architecture.

Partitioning the database — assigning tuples to buckets depending on their values for some subset of the attributes — can be used to overcome this bottleneck. This pre-processing of the data added little to the cost of inserting and deleting tuples, but allowed the relational operators to eliminate large segments of the database from consideration without having to load them into memory.

A variety of methods have been proposed for partitioning, but the one best-suited for the constraints imposed by the architecture and the problem is the RMAP partitioning. By making local partitions based on the distribution of data in a particular region, it achieves a better load factor than hashing methods, whose global strategies might result in near-empty or empty buckets.

Although the clearest gain from multi-attribute partitioning occurs when selecting or deleting tuples from the database, additional gains can be observed in relational processing by noting which buckets contain tuples that do not need to be compared...
for any given query. The sequence of partitions made to a relation forms a tree-
structure, which can be used to reduce the number of buckets that must be merged
together to form a sorted relation.

9.1 Further Directions

Real-world Database Issues This thesis focuses on the design of data structures
for rapid query execution in a parallel environment. It ignores many issues
of memory management which, although admittedly important to real-world
database management systems, are not directly related to that goal. These
topics include, but are by no means limited to, concurrency control, security,
integrity, and recover.

Algorithmic Analysis Although we were able to obtain experimental evidence that
partitioning which was sensitive to the distribution of tuples was superior to that
obtained by multi-attribute hashing methods, a better measure of the asymp-
totic performance of methods such as dynamic order-preserving partitioning is
needed.

More Complicated Split Strategies The assumption was made that all the splits
on a given level of the directory are made on the same attribute, and that the
choice of attribute is cyclically determined. But there is no inherent justifi-
cation, other than simplicity, for this approach. Favoring one set of attributes
over others will improve the performance of queries that involve those attributes,
since the granularity along that dimension will be finer. A "partition optimizer"
might monitor the sequence of queries posed to the system and impose a bias
towards those attributes which are most used in the database operators. Ac-
cordingly, attributes which are not frequently involved in queries would be given
low priority as a split attribute.
Appendix A

Data Structure Specifications

A.1 Stripes

Stripes are the basic unit of data transfer on the Connection Machine. A stripe consists of a pvar together with some auxiliary information, such as whether the stripe is present in the CM’s main memory. These operations on stripes will only function when that flag is true.

New-Stripe (&key name data-pvar) Returns a stripe which contains the data-pvar or is empty if no data-pvar is supplied.

Copy-Stripe (original-stripe &optional new-name) Returns a stripe whose data is a copy of the original stripe.

Stripe-Data (data-stripe) Returns the data stripe’s pvar.

Insert-into-Stripe (data-stripe item &optional position) Inserts the element into the data stripe at the processor whose address is position. If position is not supplied, the first null element is used.

Delete-from-Stripe (data-stripe item &key (all nil)) Deletes the item from the data stripe. If :all is true, then all instances are removed. Otherwise, only the first is.
Sort-Stripe (data-stripe &key (test '<=!!) (place-nulls :end)) Destructively sorts the elements of the stripe using the test as a comparison function. Place-nulls tells the system where to put null elements (either at the :front or at the :end).

Rank-Stripe (data-stripe &key (test '<=!!) (place-nulls :end)) Returns a new stripe whose elements are equal to the corresponding element of data-stripe's relative position in sort order.

Rearrange-Stripe (data-stripe key-stripe) Given a key-stripe, all of whose elements are non-repeating, integers or nil, and less than the number of available processors, this function permutes the data stripe. Each processor sends its value of the data-stripe to the processor indicated by the key-stripe value for that processor. If the key-stripe value is nil, then the element is discarded.

Pack-Stripe (data-stripe included-stripe) Removes all elements of the data-stripe whose corresponding value of included-stripe is null, and compacts the remaining elements into the lowest-ordered processors.

GC-Stripe (data-stripe) De-allocates the pvar containing the stripe's data.

Member-of-Stripe (data-stripe item) Returns t if some processor has the value of item for the data-stripe.

Value-Changes (data-stripe key-fields &optional (start-value)) Returns a stripe whose value is t for every processor whose value of the data-stripe is different from the preceding processor's value. If a start-value is supplied, it is compared to the contents of the first processor.

A.2 Buckets

A set of stripes which together completely specify some subset of the tuples of a relation will be referred to as a bucket.
New-Bucket (uid number-of-attributes) Returns a new, empty bucket which has number-of-attributes stripes.

Copy-Bucket (bucket) Returns a copy of the bucket.

Insert-Tuple-In-Bucket (bucket tuple) Inserts the tuple’s values into the bucket’s stripes. If the bucket is full, or if the number of attributes in the bucket is different from that in the tuple, an error is signaled.

Full-Bucket? (bucket) Returns t if the bucket is full.

Delete-Tuples-From-Bucket (bucket specifier) Deletes all the tuples in the bucket that satisfy the specifier.

Retrieve-Tuples-From-Bucket (bucket specifier) Returns all the tuples in the bucket that satisfy the specifier. Nil is returned if no tuples qualify.

Split-Bucket (bucket attribute key new-uid) A new bucket is created, which contains all the tuples in the bucket whose value for the given attribute is greater than the key. All these tuples are deleted from the original bucket.

Merge-Buckets (bucket1 bucket2) Deletes all the tuples from bucket2, and inserts them into bucket1. Returns an error if the capacity of bucket1 is not great enough, or if the two buckets have a different number of attributes.

Sort-Merge-Buckets (bucket1 bucket2 &key key-fields (test '<=!!) (place-nulls :end)) Given two buckets such that the first is sorted positively on the key fields and the second is sorted negatively on the key fields, merge the two sets of tuples so that every tuple in the first bucket is less than every bucket in the second on the key fields, and the tuples are sorted within the buckets as well.

Sort-Bucket (bucket &key key-fields (test '<=!!) (place-nulls :end)) Sorts the tuples in the bucket using the first element of the key-fields as the sort attribute. If o elements contain equal values for the first key field, then they are compared on the second. This repeats for all elements of the key-fields list. Finally,
if there are no more key-fields, and the relation has been tagged, the tag value is used.

Rank-Tuples (bucket &key key-fields (test '<=!!) (place-nulls :end)) Ranks the tuples using the sort function described above.

Tag-Bucket (bucket tag-stripe) Tags the tuples in the bucket with an auxiliary attribute, tag. This will be used to label tuples as belonging to a particular relation for binary operators, and to keep track of what processors data started out in before it got shipped around.

GC-Bucket (bucket) De-allocates all the stripes in the bucket.

Value-Changes (bucket key-fields &optional (start-value)) Returns a stripe which is true for every processor whose value of the bucket projected onto key-fields is different from the preceding processor’s.

Tag-Changes (bucket &optional (start-value)) Returns a stripe which is true for every processor whose value of the tag is different from the preceding processor’s.

Get-Median-Tuple-Value (bucket attribute) Returns the median value of all the tuples in the bucket for the given attribute.

Get-Median-of-Attribute (bucket-list attribute) Returns an integer identifying the median value for all the tuples contained in the list of buckets.

A.3 RMAP Directory

The RMAP directory is the data structure which defines the mapping of partitions to regions of the search space. It contains information on the bounds of partitions, which attribute they were last split on, and which partition they were split from. Note that
the actual buckets are not a part of the directory, so that a function which shrinks the bounds of a partition does not remove tuples from the corresponding bucket.

**New-Directory (dimensions)** Returns an empty directory.

**Copy-Directory (directory)** Copies a directory.

**Split-Partition (directory partition-id attribute value)** Given a partition ID, an attribute, and a value, split the region corresponding to that partition in two along that attribute at value. Label the region which is less than the split value with ID, and label the new region with new-ID. If the value is not in the range for the partition, do nothing.

**Merge-Partitions (directory partition-id1 partition-id2)** Modify the region corresponding to partition ID to be the union of its range and the range of partition NEW-ID. Remove partition NEW-ID from the directory. If the two regions are not adjacent, do nothing.

**Find-Partitions (directory specifier)** Given some specification of a region in the data space, return the IDs of all that partitions that overlap the region.

**Get-Sibling (directory partition-id)** Given a partition ID, return the NEW-ID of the partition which was either (1) the most recent partition split off from this one, or (2) if none exists, then the partition which split to form partition ID.

**Partitions-Overlap (directory1 partition1 partition2 attribute &optional directory2**

Given a pair of partition identifiers, tell whether or not the projection of their regions along a given attribute overlap.

**Get-Bounds (dir partition-id attribute)** Given an identifier for a partition, returns a cons cell whose car is the lower bound of the partition for the given attribute in the directory and whose cdr is the upper bound of the partition.
Get-Partition-Attribute (dir partition-id) Returns the attribute that the partition-id was last involved in a split on.

Get-Level (dir partition-id) Returns the level of the directory for the given attribute.

Get-Next-Partition-Attribute (dir partition-id &optional attribute) Returns the attribute that the partition should next be split on. If attribute is supplied, but is not one of the dimensions that the directory was defined on, an error is signaled.

Get-Next-Split-Point (dir partition-id attribute &optional value) Returns the value for the next split-attribute that the partition should next be split on. If value is supplied, but is not within the boundaries of the partition, an error is signaled.

A.4 Machine

An abstract machine has two parts: active memory and secondary storage. An active memory has a capacity, which is the maximum number of stripes that can be present at a time. Secondary storage has unlimited capacity.


Clear-Machine (machine) Takes all the stripes in active memory and moves them into secondary storage.

Reset-Machine (machine) Removes all the stripes from the machine.

Load-Stripe (machine stripe) If the active memory is not full, this loads the stripe from secondary storage into main memory. Otherwise, a warning is issued and nil is returned.
Store-Stripe (machine stripe) Removes a stripe from main memory and inserts it in secondary storage.

With-Active-Stripe ((stripe machine) &body body) A macro used to ensure that a stripe will not be stored away during execution of the body.

Make-Room (machine number-of-stripes) Makes room for the specified number of stripes by moving, if necessary, up to that number into secondary storage. If the number-of-stripes is greater than the capacity of the machine, then main memory is cleared and a warning is issued that there is not enough room for the desired number of stripes.

Reconfigure (machine new-capacity) Changes the capacity of a machine. If the change is a decrement, and main memory is overfull, stripes are moved into secondary storage until overflow no longer occurs.

Full-Machine? (machine) Returns t if the machine’s active memory is filled to capacity.

Allocate-Stripe (machine) If the machine is not full, a new stripe is allocated in main memory. Otherwise, a warning is issued and nil is returned.

A.5 Display

The display is a program framework designed for viewing the contents of stripes during the execution of *lisp functions. A machine is associated with this framework; the stripes which are in active memory are displayed in a window pane.

Com-Monitor-Stripe (stripe) Includes the stripe in the set of stripes whose contents are displayed in the window.
Com-Remove-Stripe (stripe) Removes the stripe from the active memory of the machine associated with the display.

Com-Define-Stripe (name value) Command for defining a new stripe and monitoring it as well.

Figure A-1: The RMAP display window.
Bibliography


127