BlendDB: Blending Table Layouts to Support Efficient Browsing of Relational Databases

by

Adam Marcus

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September 2008

Author .....
Department of Electrical Engineering and Computer Science
August 29, 2008

Certified by .....
Samuel R. Madden
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
David R. Karger
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

BlendDB: Blending Table Layouts to Support Efficient Browsing of Relational Databases

by

Adam Marcus

Submitted to the Department of Electrical Engineering and Computer Science
on August 29, 2008, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

The physical implementation of most relational databases follows their logical description, where each relation is stored in its own file or collection of files on disk. Such an implementation is good for queries that filter or aggregate large portions of a single table, and provides reasonable performance for queries that join many records from one table to another. It is much less ideal, however, for join queries that follow paths from a small number of tuples in one table to small collections of tuples in other tables to accumulate facts about a related collection of objects (e.g., co-authors of a particular author in a publications database), since answering such queries involves one or more random I/Os per table involved in the path. If the primary workload of a database consists of many such path queries, as is likely to be the case when supporting browsing-oriented applications, performance will be quite poor.

This thesis focuses on optimizing the performance of these kinds of path queries in a system called BlendDB, a relational database that supports on-disk co-location of tuples from different relations. To make BlendDB efficient, the thesis will propose a clustering algorithm that, given knowledge of the database workload, co-locates the tuples of multiple relations if they join along common paths. To support the claim of improved performance, the thesis will include experiments in which BlendDB provides better performance than traditional relational databases on queries against the IMDB movie dataset. Additionally, this thesis will show that BlendDB provides commensurate performance to materialized views while using less disk space, and can achieve better performance than materialized views in exchange for more disk space when users navigate between related items in the database.

Thesis Supervisor: Samuel R. Madden
Title: Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor: David R. Karger
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my advisors, without whom I would not have gotten very far in my research. Samuel Madden taught me how to think like a systems researcher, and kept me optimistic throughout our many conversations. David Karger provided me with guidance as I bumped around the many ideas we discussed, and taught me never to be sheepish about making the most of the last minute.

It was a pleasure to work with and get feedback from many bright colleagues at MIT and beyond. Specifically, I would like to thank Daniel Abadi, Sibel Adali, Michael Bernstein, Harr Chen, Alvin Cheung, James Cowling, Kate Hollenbach, George Huo, David Huynh, Harshad Kasture, Michel Kinsy, electronic Max, Daniel Myers, Debmalya Panigrahi, Vineet Sinha, Robin Stewart, Boleslaw Szymanski, Arvind Thiagarajan, Richard Tibbetts, Ben Vandiver, Eugene Wu, Mohammed Zaki, Yang Zhang, and Sacha Zyto for their discussions and emotional support. It was with you that I learned that reasonable ideas don't develop in a vacuum.

To many friends outside of school: thank you. Without you, I would not have been able to stay sane. I'd especially like to thank David Shoudy for being my partner in crime many times over, and A.J. Tibbetts for his conversation, friendship, and intertubes links. I also thank Meredith Blumenstock, who has made every aspect of my life since I met her happier, warmer, and more meaningful.

Finally, I thank my parents, who are the reason I have arrived where I stand. I dedicate this thesis to them. My mother and father prepared me for life in different ways, but throughout the time that I worked on this thesis, they both found the same way to keep me alert—constant nagging as to when I would finish. I have finally figured out the right answer: I'll call you when it's printed and signed. And yes, I am still eating my fruits and vegetables.

This work was supported by NSF and NDSEG fellowships.

Contents

Chapter 1

Introduction

Relational database systems tend to physically store tuples from the same logical table together on disk. A typical construction sorts ("clusters") the tuples of a table according to some disk-based index, and then stores the records of that table together in pages on disk. Common schema organization techniques often result in a database which stores various real-world entities (e.g., a paper, an author) in their own table (e.g., a Papers table, an Authors table).

The physical layout of a typical relational database causes the different real-world entities stored in the database to be placed far from each other on disk. Although this layout policy makes sense for many applications, there exists a large class of applications whose primary access pattern is to fetch a series of tuples from several tables to produce results for a single query. For example, in a database for a website like Amazon.com, to fetch information about a given customer's purchase history, along with information about each purchased product and some ratings or feedback about those products, it will likely be necessary to follow a four-table path from `customers.id` → `orders.customerid` → `product.id` → `ratings.productid`.

Similarly, in the "Internet Movie Database" (IMDB) [2], to show information about a given movie, including the directors, actors, and viewer ratings, it is necessary to fetch records from the `movies` table, the `actors` table, the `directors` table, and the `ratings` table. Because movies have several actors and directors, and because actors and directors work on several movies, there are additional many-to-many relationship

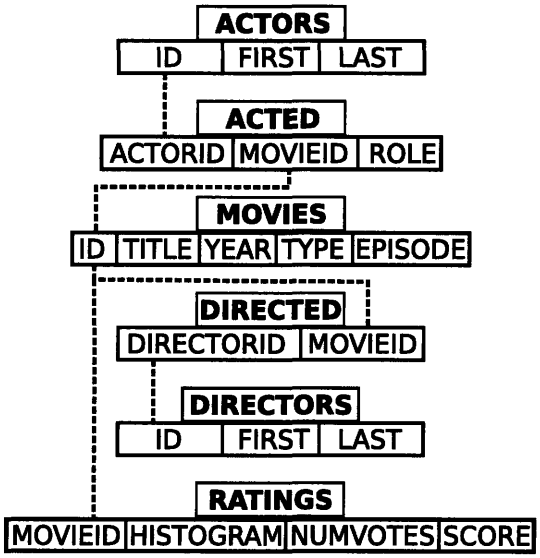


Figure 1-1: The IMDB dataset schema. Dotted lines represent key-foreign key relationships. The Acted table serves to denote many-to-many relationships between actors and movies. The Directed table serves a similar purpose for directors and movies. In a typical relational database, the Actors, Movies, Directors, and Ratings tables are stored sorted on the ID of the item they describe, with a B+ Tree index on the IDs. The Acted and Directed tables would be stored sorted on either ID type that they reference, with B+ Tree indices on both of the ID fields.

tables (such as `acted` and `directed`) involved in such queries. The schema and key-foreign key relationships for the IMDB dataset [1] are shown in Figure 1-1.

The conventional way to answer such queries would be via multiple clustered or unclustered B+Tree lookups on indices over each of these tables in search of the IDs of the desired items. Hence, to display the titles and ratings of all of the movies in which a given actor acted, the database would need to perform B+Tree lookups on the `acted` table to find the IDs of the actor's movies, followed by a series of B+Tree lookups on the `movies` and `ratings` tables to find the information about those movies, which is likely to require a large number of random disk I/Os. When a user browses to a movie listed on the page, even more B+Tree lookups and random I/Os will be needed to fetch the requested information.

Practitioners have developed several techniques to precompute or cache the results of such multi-table queries. These techniques address the multiple random I/Os required to construct one semantic entity in the database. Because the entities in

a database tend to be related (through key-foreign key references, as in Figure 1-1), further random I/Os arise as a user browses between these related items. For example, in IMDB, when a movie's actors are displayed, the user can click on a hyperlink to any of the actors for more information on that actor.

The above are typical examples of a class of workloads involving *inter-table locality of reference*, in which a query or collection of queries needs to be answered using a (often small) set of tuples from multiple tables that are all related through a variety of joins. Such workloads, which we call *browsing-style* workloads, occur commonly in many database-backed (web) browsing sessions, and share the following properties:

- They follow paths through a set of tables in a database (e.g., through a product catalog or graph of relationships), both to display a single item and to browse to related items,
- The paths they follow generally start at a primary-keyed item in the database (e.g., a movie, product, or user) and then retrieve a few records from other tables in order to fully describe the item (e.g., collecting the actor names for displaying a primary-keyed movie), resulting in a small number of tuples,
- Once a set of user queries results in information displayed to the user (perhaps in a web browser), that user often navigates to a related item in the database (e.g., by clicking on an actor that performed in the currently displayed movie), and
- They are over read-mostly data—users' queries do not insert new catalog/relationship data, and new data is typically added in bulk.

At least two general classes of activity comprise these workloads: displaying the information about a particular item in a database from records in several tables describing it, and supporting users as they browse through these linked items in a database.

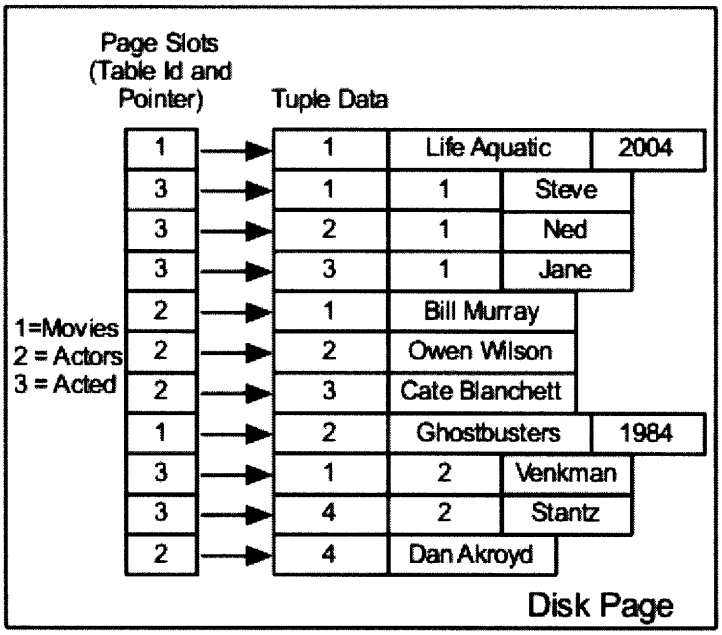
As datasets often do not fit in memory, it is likely that each of these path steps will be disk bound, because of the random I/Os incurred and the relatively limited

per-tuple processing done by the database.

In this thesis, we present BlendDB, a relational database system designed to minimize I/O costs in browsing-style queries. Rather than clustering records according to the tables in which they reside, BlendDB co-locates them with records from different tables with which they are likely to be co-accessed. The logical schema described by the user is presented in the relational interface, but the physical location of the records on disk does not follow this presentation. Our goal is to avoid multiple B+Tree and heapfile lookups when resolving a set of related queries. Thus, given a set of "paths" of interest—for example, `actors`→`movies`→`ratings`—BlendDB attempts to physically store records such that all of the records reachable along a path from a given "path head" (e.g., a particular `actorid`) are co-located together on a single physical page on disk when possible.

The "conventional" solution to this problem is to use materialized views that reduce the number of joins required by co-locating all of the desired data for a query or set of queries into one large relation. While this technique does reduce random I/Os, it has the undesirable effect of increasing data size by repeating tuple data involved on the left side of many-to-many joins. BlendDB avoids this effect by co-locating tuples that will join together, but not materializing the result of the query until query execution. An example of repeated data at the page level is found in Figure 1-2 shows a fragment of the IMDB database as it would be stored in a materialized view representation and BlendDB representation. BlendDB stores tuples that join with each other together on disk, but doesn't materialize that join; materialized views generate the result set of the join, repeating large amounts of information.

Furthermore, because BlendDB controls exactly which tuples are written to a given disk page, it has the freedom to arrange data such that all tuples needed by a set of queries that are executed together will be co-located, not just the ones needed by joins to display a single item's data. This arrangement provides space savings and reduces random I/Os further when a user navigates to a related item from their current item of interest. When humans browse a site, the path they will explore is unknown in advance, making it impossible to predict which join will need to be



BlendDB Representation

MovieID	Movie	Year	ActorID	Role	Actor
1	Life Aquatic	2004	1	Steve	Bill Murray
1	Life Aquatic	2004	2	Ned	Owen Wilson
1	Life Aquatic	2004	3	Jane	Cate Blanchett
2	Ghostbusters	1984	1	Venkman	Bill Murray
2	Ghostbusters	1984	4	Stantz	Dan Akroyd

Materialized View Representation

Figure 1-2: BlendDB and materialized view representations of a portion of the IMDB dataset. In BlendDB, every tuple is identified by a page slot, which specified which table the tuple is in. The materialized view is an entirely new table, in which tuples from the `Movies`, `Actors`, and `Acted` tables are joined together. BlendDB stores tuples that join with each other together on disk, but doesn't materialize that join; materialized views generate the result set of the join, repeating movie information.

materialized, and thus impossible to predict which materialized view to utilize for supporting a browsing session.

We will show that supporting a deeper browsing session leads to improved query throughput in exchange for an expensive disk-usage tradeoff, even in BlendDB. BlendDB's approach of co-locating all possible related objects out of a given head object aims to support any of the paths the administrator has decided a user may traverse with few I/Os. BlendDB allows a database designer to make the depth of this navigation path explicit, requiring not only the tuples needed to display one object to be available on a page, but also the tuples needed to display related objects. The disk placement of related objects suggests future work in integrating contributions from the Object-Oriented Databases community to reduce space consumption by collocating only probabilistically related objects.

In this work, we show a series of experiments on the IMDB dataset that demonstrate that BlendDB can achieve significant (up to 5.11 times as much throughput) performance gains relative to existing database systems and traditional clustering techniques. BlendDB also achieves commensurate performance with materialized views without requiring modified schemas, generation of large views to serve singular purposes, solving the view subsumption problem, or deciding which of various materialization options to utilize. Finally, we show that in the case of navigation to a related item of interest, BlendDB can achieve 1.75 times as much throughput over materialized views if we are willing to spend more disk space to support deeper navigation paths. Our solution is completely transparent to the end-user and supports general SQL queries over the blended representation.

It is worth mentioning a few caveats about our approach. First, clustering is currently performed offline. Although the system supports updates and inserts, if a large number of inserts or deletes are performed, performance may degrade and clustering may need to be re-run. Clustering is a relatively long-running operation that we imagine being run periodically, much as many existing database systems need to be periodically reorganized to promote good physical locality. Second, clustering tuples from several tables together slows down performance of queries that access a

large range of tuples from a single relation since a single table's tuples are now spread across more pages. Lastly, while BlendDB improves read performance by replicating commonly used tuples, this causes insertions, updates, and deletions to suffer, as tuple data has to be tracked and modified in multiple locations.

There are several challenges with building a system like BlendDB:

- **Clustering**: Given a collection of paths, how do we physically lay out data on disk so that we cause records originating from a path head to appear together on disk? BlendDB includes a clustering algorithm that can efficiently lay out databases according to a set of pre-specified paths of interest.

- **Replication**: Some "popular" tuples may be referenced in many paths. Should such tuples be replicated repeatedly? How can replication be exploited most effectively to improve query performance? BlendDB's storage representation allows replication of popular paths.

- **Indexing structures**: Given a "blended" representation of records on disk, how does the database system keep track of the locations of each tuple and the contents of each page? BlendDB uses a simple index to find tuples and a novel page structure to track the contents of each page.

- **Query Execution**: Within this newly created environment, how do we process queries efficiently to avoid the random I/Os seen in traditional RDBMSs on browsing-style queries? BlendDB processes most expected queries from a single disk page, incurring one random I/O as opposed to the arbitrarily many seen by traditional relational databases. The system is able to do this without increasing CPU utilization, thus increasing the throughput of such frequent queries.

- **Page Size**: Because pages in BlendDB contain more than one indexed tuple of interest, it is beneficial to increase page size if one can avoid another random I/O in the search of related information. BlendDB benefits from larger page sizes than traditional relational databases, although the desired page size is a function of hardware speed, data topology, and query workload.

Several novel contributions come out of this exploration. First is the design and development of a relational DBMS that supports inter-table clustering at the heapfile level. Second, we formalize the concept of browsing-style workloads, which justify the creation of BlendDB. Third, we introduce the concept of replication of tuple data into the well-researched field of database clustering to study how one can improve speed at the expense of disk space. Finally, we study how disk page size becomes an important factor for speed improvements in a navigational workload, where it served a smaller role when traditional clustering techniques have been utilized.

Chapter 2

Related Work

There are several well-known approaches that are typically used in database systems to cluster or co-locate tuples from related tables together to reduce the number of I/Os required to answer path-oriented queries like those we focus on in BlendDB.

We first discuss "common sense" techniques for keeping data logically separated. As a result of best practices, such as proper normalization [9], the first attempts at a schema maintain one logical entity's information in each table. Normalization techniques avoid update, insertion, and deletion anomalies [18] by discouraging repetition of information when possible. At a logical level, this design is reasonable, and allows database designers to easily maintain consistency in their data. At the physical level, however, a different organization of the data benefits throughput in the case of browsing-style workloads.

One option is to create a materialized view (MV) that contains popular path expressions. As already mentioned, materialized views can become prohibitively large. BlendDB uses far less storage by storing related tuples together on disk, but not materializing joins between those tuples, eliminating page-level and cross-page redundancy. Some databases, like Oracle, reduce redundancy by compressing commonly repeated tuple data at the page level, but if the materialization of a query grows past a page boundary, replication across pages cannot be controlled [17]. It is not hard to imagine how a materialized view that, for example, joins movies to all of their actors, directors, and ratings, might record many pages of information about a given

movie, making such compression difficult or impossible. It is also not sensible to first materialize the data required for a query and then compress it, rather than avoid the physical materialization in the first place, as BlendDB does.

Materialized views also add overhead in forcing the user or system to rewrite queries [8] to use tables other than those of the original schema, and impose complexity if attempting to solve the difficult problem of determining which materialized views can answer a query [11, 14]. Furthermore, as discussed above, views can only be materialized if the necessary joins can be known in advance. A user browsing through data will effectively generate a lengthy path that can only be known at the end of the browsing activity, preventing us from materializing the join in time.

Relational databases as early as System R [6] provided a notion of "links" between tuples related by joins in the storage layer, but this was used to avoid index lookups while still jumping to a different page to retrieve a joined tuple. Tuples from different pages could be co-located, but this functionality was not exposed in the Relational Data Interface, nor could we find a performance evaluation of the technique.

Bitmapped join indices [16] are commonly used in star-schema database systems. If a table that is low-cardinality (has a small number of rows) is joined with a table of high-cardinality, one can perform the low-cardinality join first, and utilize a bitmap index of the joined locations in the high-cardinality table to filter the result set. While such optimizations improve performance significantly for these schemas, navigation-oriented queries of the kind we study do not feature low-cardinality tables on either side of a join, and so join indices do not benefit our schemas.

A recent paper describes the design of merged indices [12] for relational databases. These indices allow keys from various columns and tables to be interleaved in a structure similar to a B+ Tree, thus allowing two records from different tables to be co-located in the index if their indexed values are the same or similar. If the heapfile is clustered on this index, the result is similar to that of BlendDB for the case of reducing the random I/O's required to materialize an object. BlendDB extends on this work by considering how to further benefit from a read-mostly workload, namely by showing that repetition of records at the physical level can improve performance.

Additionally, the work described only applies when multiple tables are co-accessed through a join relationship on their key values, as applies in the case of rendering an object a user wishes to see. The proposed design does not cover the browsing scenario, where a differently keyed object is rendered for the user.

Clustering related records is a topic as old as hierarchical databases such as IMS. Followed by the network model, as embodied in the CODASYL consortium of the 1960's and 1970's, these two models described how to store databases for tree- and graph-strucutred data. The natural idea to phyiscally co-locate items that are traversed by following relationships fell naturally out of these models. How to carry these physical designs to a relational model is not immediately clear in the face of many-to-many relationships, and it was not until the object-oriented database literature described algorithms for disk layout that the field matured.

Clustering algorithms that place many objects into a single heap file appeared quite frequently in the OODBMS literature. For example, the O2 DBMS [7] allowed related objects to be clustered together. Tsangaris and Naughton [22] provide a comprehensive study of various object-oriented database clustering algorithms [20, 10, 21], and these techniques can be used as clustering algorithms in BlendDB. We believe we are the first to propose applying such clustering to relational databases. The algorithm that we propose in this thesis does not build on the statistical methods explored in the OODBMS literature, and thus suffers from increased heap file size that could benefit from statistical pruning. As it is possible to reduce heap file size by applying statistical techniques when user access data is available, we do not explore this route further, as it an obvious engineering effort to extend our own work.

Being able to support general relational queries on top of a blended database requires significant additional machinery in BlendDB, as objects are not explicitly defined. Furthermore, BlendDB's clustering algorithm allows information about a given "object" (e.g., a movie) to be replicated in a few locations, something that has not been explored in the literature. This replication can further improve performance and reduce the complexity of a clustering algorithm by avoiding the constraint satisfaction problem that arises once an object is restricted to one location on disk. We do

not claim that the techniques for performing such clustering in relational databases are significantly different (other than allowing replicated data) than those found in the OODBMS literature, nor do we wish to propose clustering algorithms that outperform those techniques, but instead wish to explore these notions in a relational context. We feel that such a context is one that better applies to today's popular web-oriented database systems, which primarily utilize relational technology.

Some commercial systems provide "multi-table indices" that allow indices to be built over several tables; for example, "Oracle Clusters" [4, 13]. Such indices can be physically clustered so that records from different tables are in fact co-located in the same page on disk. These work by mapping records with the same key value (e.g., movieid) to the same page on disk, with one or more pages dedicated to each key. Different keys are always on different pages, potentially wasting substantial space (Oracle documentation [4] recommends "A good cluster key has enough unique values so that the group of rows corresponding to each key value fills approximately one data block. Having too few rows for each cluster key value can waste space and result in negligible performance gains.") Paths representing a series of joins cannot be materialized through clusters. Finally, the replication of tuples for performance is not considered. BlendDB addresses all of these concerns.

There has been work in the XML community on path indexing in semistructured data, as was done in the Lore project [15]. Lore focuses on providing unclustered indices that make it possible to follow a path, but does not discuss physically co-locating records.

Observing what practitioners in the field do to avoid random I/Os helps guide our work. One of the most popular approaches in the web community for high-throughput queries is to cache requests for objects in the database in a scalable way. A common technique is to use *memcached* [3], a distributed LRU cache that presents a key-value lookup interface. Layering such a system on top of a database reduces the random I/Os associated with constructing frequently accessed items, but still incurs the costs of "warming" the cache up and answering long-tail queries. Techniques to physically reduce the random exploration of the heapfile, such as the ones employed

by BlendDB, can be used in conjunction with caching layers to reduce load on the server which ultimately stores the data.

Chapter 3

System Overview

BlendDB system consists of two components: a clustering algorithm, and a query processor. Tables are processed by the clustering algorithm and written to a single "blended" heapfile which contains the tuples from the various tables mixed together for optimized disk access. The query processor operates on this blended representation.

3.1 Preliminaries

We begin by defining to concepts necessary for describing the overall architecture. Because our concerns are primarily in retrieving tuples by a series of key/foreign key joins, we need to reason about groups of tuples from a table with the same key value. This concept is known as a key group.

Definition 1 *A* Key Group $K = K_{R,f,v}$ *is a set of tuples from relation* R *with a value of* v *in field* $R.f$. *If the field* $R.f$ *is a primary key of* R, *then* $|K_{R,f,v}|$[1] $\in \{0, 1\}$. *Otherwise,* $|K_{R,f,v}| \geq 0$, *as is the case if* $R.f$ *is a foreign key reference.*

For example, the key group $K_{actors,id,1}$ is a set containing one tuple with the actor having a primary key (id) of 1. The key group $K_{acted,actorid,1}$ is the set of tuples representing the movies that actor 1 acted in.

[1] *For a set* A, $|A|$ *denotes the number of elements in* A.

We now want to consider the collection of tuples that can be formed by starting from an index lookup to some key group in relation R_1 and performing a series of key/foreign key joins through relations $R_2, \ldots, R_n$. In browsing-style queries, one generally starts at a given keyed record in a table, and performs a series of key/foreign key joins resulting in a relatively small set of records. This path of joins through tables is known as a *join path.*

Definition 2 *A* Join Path *is a set of tuples defined by an initial key group* K_{R_1,fin_1,vin_1} *used for a selection into a table, and a sequence of relations* $(R_1, fin_1, fout_1) \rightarrow \ldots \rightarrow (R_n, fin_n, \emptyset)$. *An item i in the sequence contains a relation* R_i, *a field* $R_i.fin_i$ *which joins with* $R_{i-1}.fout_{i-1}$ *(the previous sequence item), and a field* $R_i.fout_i$ *which joins with* $R_{i+1}.fin_{i+1}$ *(the next sequence item).*

We refer to the initial key group as the **head** of the path.

A trivial example of a join path is a selection on the movies table for movie id 5. This is represented by a path with key group $K_{movies,id,5}$, and sequence $(movies, id, \emptyset)$. A slightly more complicated path that evaluates the actors in movie 5 would be represented by key group $K_{acted,movieid,5}$, and the sequence $(acted, movieid, actorid) \rightarrow (actors, id, \emptyset)$. For the most part, multiple such join paths arise in rendering a single object, but there is no restriction dictating that the join path can't include tuples arrived at by browsing a join path to a related object.

Finally, we recognize that a particular action—such as rendering a page of data about a movie—may require performing several distinct queries about an item with the same key value.

Definition 3 *A* Join Path Set *is a set of join paths that all share the same initial value for their key group.*

The join path set for a movie would include the four join paths required to display its name, actors, directors, and ratings. All four of these paths originate at different tables, but begin with a selection on the same movie id value in each of those tables. Note that a join path set does not necessarily require that all of its join paths have

the same *key group* at its head—it only requires that all of the key groups at the head of each path have the same *value*. For example, in the join path set for a movie, the movie data (through the `Movies` table), actor data (through the `Acted` table), director data (through the `Directed` table), and ratings data (through the `Ratings` table) are all keyed on a given movie id, but do not have the same key group at their head, as a key group specifies a table in its definition.

3.2 System Operation

Because we are interested in using BlendDB to improve the throughput of queries with many key-foreign key joins, for the remainder of the thesis, we will only consider equality joins.

When a SQL query arrives, it is passed to the query processor for execution. First, the query optimizer tries to determine if the query can be answered by a join path by looking for a path head bound by a value (e.g., $movies.id = 5$). If this is the case, the head of the path is looked up in the *key group index*, which determines which page to read for that key group. Ideally, this page will contain the entirety of the join path for the path head. Hence, when the join to the next column in the path is performed, BlendDB searches pages that have already been accessed for this query for the desired key groups at the next step. If a key group is not found in the current set of pages, BlendDB searches the appropriate key group index and retrieves the key group from a new page. Once the join path has been traversed, tuples for the result set are materialized and returned to the user. We describe several techniques to support generic relational queries in Section 4, but emphasize that BlendDB is unlikely to perform well for workloads that do not primarily contain join path queries.

We discuss the details of the clustering algorithm in Section 5. The goal of the clustering algorithm is to write each join path set to one page. Ideally, we write the results of all queries that are issued at the same time about a given entity with the same id to the same page.

As an example, consider the four join paths originating at a single movie. The

database administrator creates this join path set, as well as similar ones that define the queries to fully describe actors and directors. These path sets are then passed as arguments to the clustering algorithm. In order to show the movie's title, actors, directors, and ratings, we would co-locate all tuples along the paths below that have the same integer movie identifier at their head:

{`movies.movieid`},

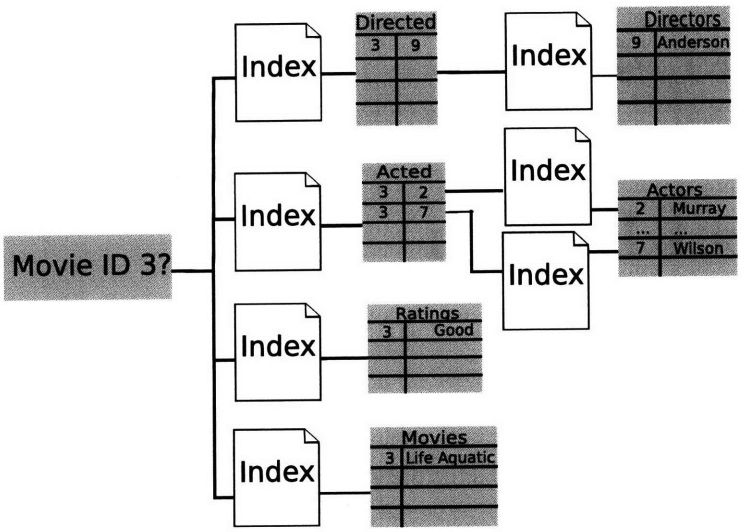
{`acted.movieid`→`actors.id`},

{`directed.movieid`→`directors.id`}, and

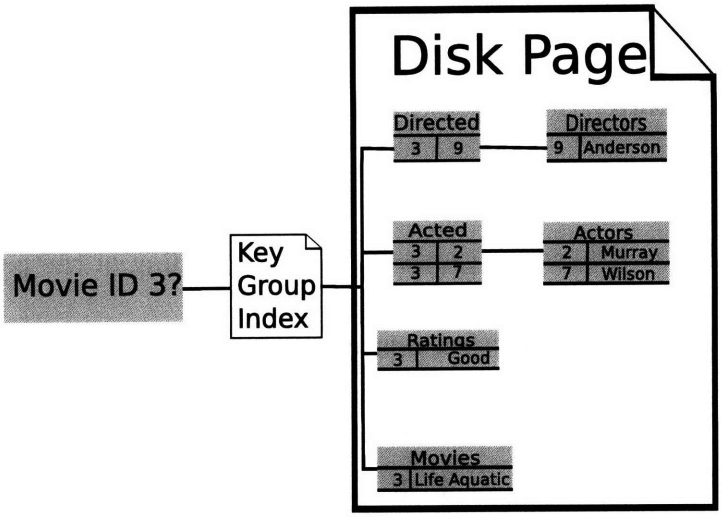
{`ratings.movieid`}.

The clustering algorithm does this by starting with a particular path head, following all of the join paths out of it (via key-foreign key joins), and writing out the joining tuples to the same disk page. It does this for every path in a join path set, so that all of the queries for a given head with the same integer id can be answered from the same page.

Figure 3-1 shows the pages and I/Os incurred by BlendDB and a traditional RDBMS such as Postgres in answering those queries. Note that we perform a single key group index lookup to find the appropriate page for the heads of the four path queries, and then are fortunate enough to find all subsequent key groups on the same page, resulting in a total of two random I/Os. In the case of the traditional RDBMS, complex join algorithms are not used because of the small amount of resulting tuples. As such, an index nested loop join is performed for each table joined, and so an I/O per index and an I/O or more per table is incurred to answer the query. The scenario described for BlendDB is optimal, but note that if some of the key groups were not located on the current page in executing these four queries, BlendDB's query plan would gracefully degenerate into that of the traditional RDBMS—it would consult a separate index for each desired key group, and the disk would have to seek to another page to retrieve that key group, incurring two I/O's for each key group—the amount required by a clustered traditional RDBMS. It is also possible that a key group have too many tuples to fit on a single page—in this scenario, overflow pages ensure correctness at the same performance cost witnessed by a traditional RDBMS.



(a) Four related query paths processed by Postgres



(b) Four related query paths processed by BlendDB

Figure 3-1: The same four queries for a movie's title, actors, director, and ratings as accessed by (a) Postgres and (b) BlendDB. Postgres performs random I/Os in six tables and six indices. BlendDB seeks to a Key Group index for movie id 3, finds the appropriate page, and seeks to that, resulting in two random I/Os.

Chapter 4

Physical Design

We now describe the physical design of BlendDB files, including page layout and indices. We discuss the details of the clustering algorithm in Section 5.

4.1 Data Page Format

Pages are modeled after the description provided by Ramakrishnan and Gehrke [18] with a few modifications. In short, a data page contains the following:

- The tuple data list, which grows from the beginning of the page,
- A region of unused space (free space) that the tuples and page slots grow into,
- The page slot list, which grows backward from the end of the page, and specifies what exists in various parts of the tuple data region,
- The free space byte counter (this is the difference between the end of the slot list and the end of the tuple data list),
- The number of slots found on the page
- A next page pointer to a spillover page, for scenarios where key groups take more than one page

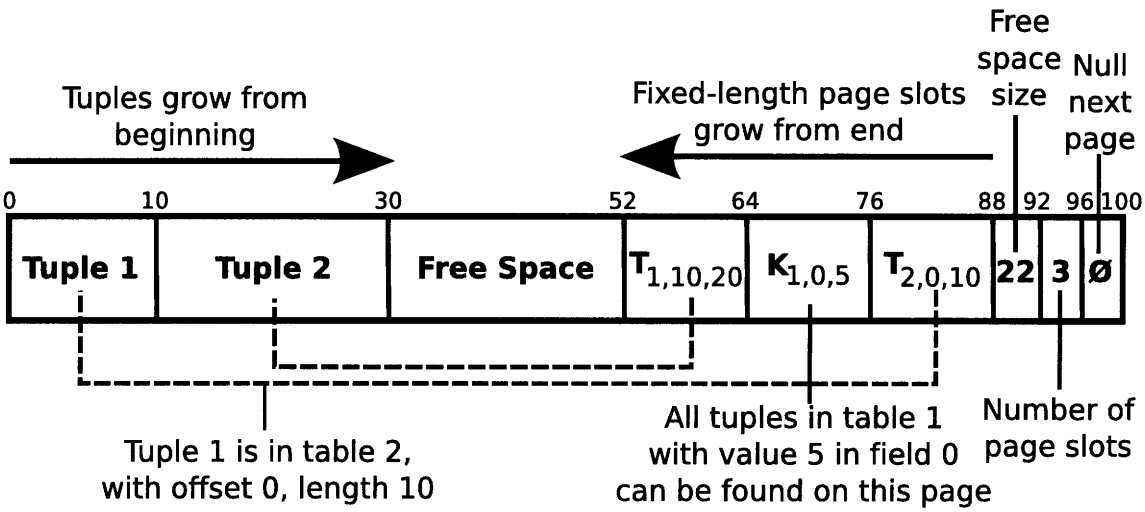


Figure 4-1: Example BlendDB page with two tuples. The page is 100 bytes, which is small for educational purposes. Page sizes are usually a power of two. It has three page slots, two of which contain pointers to tuples. The other slot indicates that the key group tuples for relation 1, field 0, and value 5 are fully contained on this page. The page has 22 bytes of free space, and has a null next page pointer.

An example of the page format can be seen in Figure 4-1. There are several differences between Ramakrishnan and Gehrke's page format and the one that BlendDB uses. First, fixed-length tuples are no longer an option, because tuples from different schemas are stored on the same page. As such, all tuple data is variable length. Second, because a heapfile in BlendDB stores multiple relations, a page slot must identify the table that a tuple belongs to. Finally, in order for the query processor to take advantage of a page that contains all of the tuples of a key group, this fact must be denoted on the page.

All page slots are fixed-length, and contain three integer values. Page slots are stored in a list, and can describe one of two types of items on the page. The first bit of the ith slot determines the type of information in that slot:

- If it is 0, then the following 31 bits represent the table a tuple is from. The next two integers represent the offset and length on the page.
- If it is 1, then the following 31 bits represent the table ID of a key group. The next two integers represent the field id and value of the column that the key group represents. This means that all of the tuples of the described key group

can be found on this page.

While our focus is on key-foreign key joins and lookups, if a user wants to build a key group on a non-integer field, BlendDB can create a dictionary encoding table (not visible to the user) that maps unique non-integer objects to integers at the (typically) in-memory expense of processing the extra dictionary lookup to the same page.

If a page contains an empty next page pointer, then it has no overflow information. The page pointer is only used when a single key group is written to the page and takes more space than a single page. In this case, the page pointer will contain the page number of the next page that contains the remaining key group tuples. Each page in this linked list of pages will contain a single key group entry in all of its page slots, and contain tuple offset information for tuples in that join group in the other page slots. To avoid ambiguity, one can not begin to write a key group to a page and end up writing the remaining tuples for the key group to another page unless there are no other key groups on the page, and the key group fills more than one empty page.

4.2 Tuple Format

BlendDB supports variable-length tuples. In on-page binary form, a tuple is represented by writing its fields contiguously to disk. For fixed-length fields, only the field data is written. For variable-length fields such as strings, a 4-byte integer length is prepended to the field. It is thus possible to read a field from this binary representation without processing the other fields, which is a useful property that BlendDB exploits as it searches for a key group on a page.

4.3 Key Group Indices

BlendDB supports standard B+Tree indices over any field. These indices map from a given value v of the indexed field to a list of pointers to page slots that contain tuples with value v in that field. These indices serve the same purpose (supporting

selections and joins) and act as traditional indexing techniques in RDBMSs. Because tuples can be duplicated on multiple pages in BlendDB, clustering algorithms are prevented from adding a tuple's location to the index twice by maintaining a bitset of tuples not yet indexed.

A *key group index* (used in join path query execution) is also a B+Tree, but rather than mapping from an arbitrary field value to the page slots containing that field value, the key group index for a given field maps from an integer value on that field to a page that contains the key group (the page that contains its first few tuples if it spills to more than one page). If a key group is too large to fit on a page, it overflows to a new page. This is indicated on the page pointer of the original page in order to keep the key group index compact. Because it maintains a compact mapping, it is more likely that a key group index will fit in memory than a traditional B+ Tree index, at the expense of having to scan every tuple on the desired page to retrieve all tuples in that key group. This index does not include a list of all pages on which a key group is found—it only stores the "best" page on which that key group appears, as described in Section 5. To support insertions, updates, and deletions in the face of replication, we discuss how to modify or complement the key group index in Section 6.

4.4 Sequential Scans

Aside from index lookups, we must also support sequential table scans in order to fully support arbitrary query plans. While performance of a scan of a table is degraded due to a blended heapfile, we provide support by maintaining a sequential index. This index is built by performing a single scan of the heapfile, and logging the page slot of the first instance of every tuple. Once all tuples are accounted for, a sequential scan of a table can be implemented by reading the sequential index and heapfile in order, and constructing tuples from the page slots that are specified in the index. If the random I/Os experienced during this pseudo-scan of the table are unacceptable, a separate replication of each tuple of the table can be placed on disk in sequence, and the index can point to this copy for improved scan efficiency at the expense of

join path evaluation performance.

4.5 Buffer Manager

The buffer manager handles loading pages from disk and retiring them on a least recently used basis. Initially, we designed BlendDB to add a mapping for each key group on a loaded page to a global hash table of in-memory key groups so that subsequent lookups in join paths would not have to consult the key group index and potentially read a new page from disk. The in-memory hash table we used caused too much memory allocation and deallocation for each key group on a page that entered or left the buffer pool. As we searched for an alternative, we found that a brute force sequential scan of a per-query buffer pool (a *page set*, described below) did not significantly affect CPU utilization for the page sizes we use. Hence, this is the approach we use.

4.6 Processing a Query

We now have all of the components required by our system to describe how queries are processed in BlendDB. We will restrict our discussion to the evaluation of a join path starting at key group K_{R_1,fin_1,vin_1} and a sequence of joins described by $(R_1, fin_1, fout_1) \rightarrow ... \rightarrow (R_n, fin_n, \emptyset)$.

Because join paths contain one selection and a sequence of small joins, a reasonable query plan heuristic for join path queries is a left-deep plan implemented by a series of index-nested loop joins, as seen in Figure 4-2. We push the selection of vin_1 on $R_1.fin_1$ to the base of the plan, and then perform the joins from left to right as they appear in sequence. While these joins are inexpensive on their own, they traditionally result in at least as many random I/O's as there are tables along the path being joined, which results in poor throughput if many such queries are run concurrently.

We must first define one notion to understand how BlendDB evaluates this plan.

Definition 4 *A* page set *for a query is the set of pages that have already been accessed*

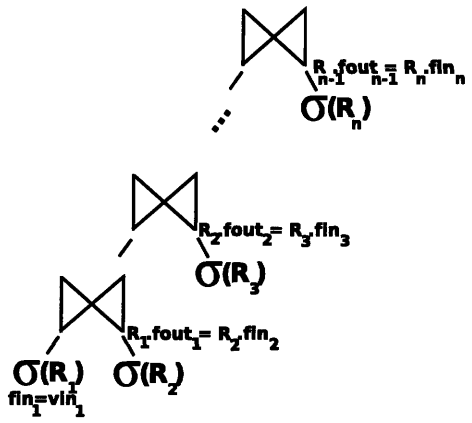


Figure 4-2: Typical query plan for implementing join path queries. A single selection on R_1's key group K_{R_1,fin_1,vin_1}, and then joins between $R_i.fout_i = R_{i+1}.fin_{i+1}$.

as part of evaluating the operations lower in the query plan.

Each query is assigned to a new page set. As a hint, the system can assign a query to the page set of a previous one. This is done when queries are expected to be related, and thus the tuples satisfying the query co-located. For example, a query for the actors in a movie and the rating of a movie should be co-located, and placing those two queries in the same page set will allow them to share pages without accessing the general buffer pool. This allows us to maintain a list of pages to search for desired key groups on before having to consult the key group index and potentially incur more random I/Os.

The plan is executed as follows by the standard iterator model of query plan evaluation:

1. We consult the page set in case a previous query in our page set has loaded a page which answers our query already. For all pages in our page set, we perform a sequential scan of the page slots, looking for a record indicating that K_{R_1,fin_1,vin_1} is located on this page. We have found that this scan is negligible compared to the random I/Os required to consult the key group index and load a new page.

2. If the key group is contained on the page, we scan the tuple data to find all tuples that are in this key group. If it is not, we consult the key group index

for $R_1.fin_1$ to locate the "best" page containing vin_1. We load this page into the buffer manager, add this page to our page set, and retrieve the desired key group.

3. Given the tuples discovered at level i of the path, we look at field $R_i.fout_i$ of each tuple, get its value $vout_i$. We search for $K_{R_{i+1},fin_{i+1},vout_i}$ within the current page set by a sequential scan of its pages. If it is found, we load its tuples, and if it is not, we load the page after consulting the key group index.

4. We repeat step 3 for each tuple at level $i+1$ until we have covered all n steps of the join path sequence.

This process is essentially an index nested loops join, with two differences: the index used is a key group index, and for a well-clustered heapfile, the index will not be consulted after the first page is loaded into the page set, as all subsequently desired key groups will be co-located on the same page.

Chapter 5

Clustering Algorithm

We now describe the clustering algorithm that lays out key groups on disk. We present two algorithm variants: the first retrieves key groups by performing a random I/O per group. This algorithm provides good guarantees about key group layout, but the approach results in scalability concerns once the dataset being clustered no longer fits in memory. The second replaces these random I/Os with sequential scans that retrieve multiple key groups at once, allowing it to scale to much larger datasets. We then describe modifications to the algorithm to improve the throughput of naviational browsing patterns. We end with a modification to decrease the disk space utilization of the clustering algorithm by suppressing the frequent repetition of larger key groups.

5.1 Basic Clustering Algorithm

At a high level, the clustering algorithm we describe ensures that for any instance of a join path set that an administrator defines, we will write that entire join path set instance (the set of tuples accessible by the join path set with a given id for its path heads) to the same disk page. We only fail at this guarantee if the set does not fit on a new page, as nothing else could be done to fit this set on a page. If a set is written to a page and more space remains, we pick a set that shares tuples with one already written to a page, and attempt to add that set to the same page.

Algorithm 1 shows the basic outline of the clustering algorithm. This is a simplified

Algorithm 1 Clustering Pseudocode

```
1: while remaining_heads_queue.size() > 0 do
2:    next = remaining_heads_queue.pop()
3:    cluster-head(next, heapfile.newPage())
4: end while
5:
6: function cluster-head(head_keygroup, page):
7: heads.push(head)
8: while page.hasSpace() do
9:    if heads.size() != 0 then
10:      follow-path-forward(heads.pop(), page, heads)
11:   end if
12: end while
13: for all written heads w ∈ page do
14:   if head == w OR satisfaction-check(w, page) then
15:      index(w, retrieve-kg(w))
16:      remaining_heads_queue.remove(w)
17:   end if
18: end for
19:
20: function follow-path-forward(head, page, heads)
21: for all paths p ∈ head.relatedPaths() do
22:   next_kgs.push(p.head)
23:   for all (R, fin, fout) ∈ p.sequence do
24:      current_kgs = next_kgs, next_kgs = ∅
25:      for all keygroups kg ∈ current_kgs do
26:         tuples = retrieve-kg(kg)
27:         status = write-kg(kg, tuples, page)
28:         if status == fit_on_page then
29:            p.head.satisfied = true
30:            if kg != p.head then
31:               index(kg, tuples)
32:               if kg.isHead() then
33:                  heads.push(kg)
34:               end if
35:            end if
36:         else
37:            p.head.satisfied = false
38:         end if
39:         (Rnext, finnext, foutnext) = next step in p.sequence
40:         for all tuples t ∈ tuples do
41:            joined_kgs = get-joined-kgs(tuple, fout, Rnext, finnext)
42:            next_kgs.push(joined_kgs)
43:         end for
44:      end for
45:   end for
46: end for
```

version of the implementation; where appropriate, we describe major features not shown in the pseudocode in our description below.

We first describe the inputs to the algorithm. We assume that the tables being loaded are provided to the algorithm as relations on disk, and pre-processed so that temporary (secondary B+Tree) indices are created on each key group column, allowing key groups to be retrieved as needed by the clustering algorithm. We also assume that the database administrator provides us with descriptions of all join path sets they expect the system to encounter.

The algorithm begins in lines 1-4, by writing any path head not yet *satisfied* to a new page. A path head is satisfied once all paths from it have been written to a single disk page. If we can not fit the paths out of a head on a page which we explicitly started for that head, then we have made a best effort, and consider it satisfied regardless. `remaining_heads_queue` is implemented by maintaining a hash table of path heads yet to be clustered, and a priority queue of these heads. The queue is initialized with all path heads not satisfied on disk. Priority is assigned to path heads based on how many times they have been written to various pages without being satisfied – this happens for popular heads which are too large to fit on a page and otherwise litter other pages without being satisfied.

On line 2, we pop an unsatisfied path head off of the priority queue and store it in `next`. We have found that this priority queue cuts the heapfile size in half for the IMDB dataset compared to random path head selection. Once we have selected a head to start a page, we begin the clustering process with the `cluster-head` function on lines 6-18. Here we maintain a list of path heads that we have written to disk while clustering the current disk page. Determining that a key group is a path head is a simple operation—if the key group's table and field define the head of some path in the desired list of join path sets to cluster, then the key group just written to disk is the path head involved in some join path set. The purpose for this step is two-fold. First, we write paths that have heads discovered by the clustering algorithm to the same page when possible because it saves space—for example, if we have written all of the actors, directors, and ratings for a movie to a disk page, we may as well reuse

the written actor and director tuples to write all of their respective movies, instead of repeating them elsewhere. Second, writing these discovered path heads to the same page is what allows us to answer navigational queries more efficiently. If a user is browsing an IMDB entry for a movie, sees an actor that is interesting and clicks on that actor, we want to answer the subsequent queries for that actor from the same disk page whenever possible.

The heart of the algorithm is contained in the `follow-path-forward` subroutine. The first line of this function retrieves all join paths that the administrator has told us are contained in the join path set containing `head`. If the path head we start with is $K_{actors,id,3}$ with join sequence $\{\texttt{actors.id} \rightarrow \emptyset\}$, then the other join path instance in its join path set will be defined by path head $K_{acted,actorid,3}$ and join sequence $\{\texttt{acted.actorid} \rightarrow \texttt{movies}\}$. When possible, these two paths should be written to the same page. For each path, we attempt to write all key groups starting at the paths' heads, and move through the sequence of joins for each path from left to right.

For any given path, we loop through all of the steps (key groups) of its join sequence starting on line 23. The `retrieve-kg` function call on line 26 retrieves the set of tuples involved in a key group. The `write-kg` function call on line 27 handles the logic of only writing a tuple to a page once, even if it is involved in two key groups written to the same page, and returns whether there was enough room on the page to fit the key group. Assuming it fits (line 28), we index the tuples by updating the appropriate key group indices for key group locations, and non-key group indices for tuple locations. For any key groups along paths that are heads of a path other than the current head, we add those to the list of discovered heads. If the key group in this step of the path does not fit due to the page being full, we mark the current path head as not satisfied, as we have not written its contiguous join sequence to a disk page. In lines 39–43, we perform the join from path level i to path level $i+1$, thus populating the list of key groups to retrieve for the next level. While not written explicitly, we support logic to stop whenever a page fills up, and can take an argument telling the algorithm the fill factor of a page, a traditional concept for how much space to leave for future insertions.

Once a page has filled up, we return to lines 13–18, where we loop through all path heads written to the page. The call to `satisfaction-check` on line 14 verifies that all paths in the join path set of a path head have been satisifed (written to the same page). In this case, the page is considered the authoritiative container of the path head, which is indexed to denote this.

5.2 Scaling to Larger Datasets

There are two parts of the above algorithm that do not allow it to scale to larger datasets. The first is that `retrieve-kg` performs a random I/O for each key group in each explored path if the dataset is not memory-resident (we are essentially performing every possible join in the system, some more than once!). The second is that building the index by calling `index-kg` inserts data into the B+Trees out of order, resulting in unscalable insert performance for larger indices. This second issue can be addressed easily – we keep a log of which index entries we need to update, wait until the end of the clustering, sort the log by B+Tree key index, and perform the insertions in order. The first issue requires further attention.

We provide a rough outline for how to proceed with a more scalable solution to retrieving key groups. Instead of retrieving key groups in an ad-hoc manner, we retrieve them by sequentially scanning the relation we desire to retrieve key groups from. In order to not perform one sequential scan per step in the path, we "parallelize" requests for these key groups. This is accomplished by placing one path head per page for a large set of pages that can fit in memory. Instead of calling `retrieve-kg`, we form a hash table of key groups that we wish to retrieve, and register a callback for when that key group is scanned from disk, similar to what is done in a simple hash join described by Shapiro [19]. Next, we sequentially scan the desired relation, calling the registered callback on any page which should contain a requested key group. We then process the logic in the original algorithm on the tuples from that key group. This will lead to other desired key groups, which we will process through further sequential scans.

5.3 Supporting Navigation by Clustering Longer Paths

The clustering algorithm, as described, enforces that sets of paths out of a key group are written to the same page. This can be used to join several queries related to an item that may be displayed to the user at once. In order to support navigation to related items, much of the prior work in the OODBMS clustering field [22] has focused on collecting browsing statistics of co-accessed items, and writing these co-accessed items to the same disk page. Such statistics require bootstrapping and instrumentation of a working system, which requires access to information that is not always available, as was our experience with evaluating the IMDB dataset.

One way to make the clustering algorithm support navigation between items in the database is to define longer join paths in join path sets. For example, the paths in the join path set for displaying actors are:

{`actors.id`} and

{`acted.actorid`→`movies.id`}.

and the paths for displaying a movies are:

{`movies.id`},

{`acted.movieid`→`actors.id`},

{`directed.movieid`→`directors.id`} and

{`ratings.movieid`}.

If these were the only paths the administrator provided the clustering algorithm, then all actor-related information for any actor would appear on a single disk page when possible, as would all movie-related information for each movie. To cluster queries displaying all movie-related information for movies related to an actor on the same page as the information for that actor, the administrator can modify the paths for displaying an actor's information:

{`actors.id`},

{`acted.actorid`→`movies.id`},

{`acted.actorid`→`acted.movieid`→`actors.id`},

{acted.actorid→directed.movieid→directors.id} and

{acted.actorid→ratings.movieid}.

Essentially, the adminsitrator must combine the set of paths required to display a single object described in the database with the paths required to display an object related to it. This technique provides the database administrator with fine control over which paths through the tables to support, but is cumbersome in that its description length grows exponentially in the depth of the navigation supported. The technique requires no additional facilities on BlendDB's behalf to support clustering deeper navigation, but the semantics of the technique are difficult to work with.

Given that the clustering algorithm we describe does not take statistical access data into account, we we propose adding another argument to the original clustering algorithm: path depth. An adminstrator does not have to construct paths to support navigation unless she wants to—instead, she can provide the join path sets to display individual objects, and also request that the algorithm cluster objects which are a navigation depth d away to the same page.

The original algorithm already attempts to write a path head that is discovered while writing a related path to the same page in order to save space. Currently, the call to `satisfaction-check` on line 14 considers a key group satisfied if all of the paths in its join path set have been successfully written to the same disk page. Consider instead requiring `satisfaction-check` to mark a key group satisfied only if the tuples along its join path set are written to disk, as well as those of all path heads discovered while writing its join paths to disk up to a depth d away. In this case, a user could navigate to a related item up to d items deep without falling off of the page used to display the original item.

With this notion of path depth in mind, we see that the original clustering algorithm clusters with a path depth of 0 – only the immediately requested item is guaranteed to be found on a single disk page. In Section 7.4.4, we explore the tradeoff between the space required to store this precalculated exploration and the throughput improvements of reduced random disk I/O when increasing path depth. The space utilization can be lowered by utilizing one of the statistical methods discussed

in Section 2, should such additional data be available.

5.4 Preventing Repeated Large Path Heads

Some key groups are quite large – the most popular actor has 5046 entries in the `acted` table. These popular key groups tend to be on many join paths in the dataset, yet many just barely fit on a page on their own. Once such a key group is pulled onto a page by a join path, it prevents many smaller key groups from satisfying paths on which they exist. Additionally, a key group so large that it fits on its own page would require a random I/O regardless of how we cluster, so we may as well leave it out of clustering to allow space for key groups for which we can avoid extra I/Os.

We prefer to supress such key groups from being written to many pages and preventing the successful path traversal of the other queries answerable on a given page. In line 2, we select the next head to write to a new page. Since `remaining_heads_queue` is prioritized by the number of times a path head has been written to a page without being satisfied, the most frequent page-filling key groups tend to rise to the top of this queue. We select a threshold, which is currently set to 2 previous unsucessful appearances, for a key group to be added to a list of suppressed heads after being written to its own page when popped off the queue.

Once on the suppressed heads list, a key group that is discovered while writing a path to a page is prevented from being written to that page, in order to allow other smaller join path sets to be written to the page. This results in significant space-savings (greater than twofold savings), and improved performance, as more paths traversed are located on a single page, while larger, frequently accessed key groups will appear in cache in a high-throughput scenario.

There is one downside to this approach – it is biased toward smaller page sizes. We can only select one path head to suppress per page, since we can only be guaranteed to satisfactorily cluster the first join path set written to a page. A clustering for smaller page sizes will write more pages to disk, and thus react to offending key groups more quickly, as more of them will be suppressed with less wasted space. This situation

arises more noticably for path depth clusterings greater than a depth of 0, as more tuples are required to exist on a page for a path head to be considered satisfied, and thus more offending key groups must be suppressed. We see this effect in the experiments of Section 7.4.4. A possible solution to this issue would be to prioritize suppressed key groups *a priori*. Before clustering begins, we can scan the key group listings, calculating the size of each. We can then select some small percent of the largest key groups or join paths to suppress. Clustering can write these key groups to disk first, and then proceed with the original algorithm.

Chapter 6

Insertions, Deletions, and Updates

In this section, we explain how BlendDB handles insertions, deletions, and updates. We note that these operations cause data to become clustered in a non-ideal manner over time, causing join paths to be spread across pages. Hence, periodic reclustering may be needed if the operations are very frequent.

As discussed in Section 4.3, the key group indices used for querying only contain the optimal page to find a key group on. Due to replication, a key group can exist on several pages, and all of these pages must be accessed in the face of updates, insertions, and deletions. To support a read-write model instead of a read-only one, a key group index must store all pages that contain a key group. We store this *complete* key group index separately from the one used for query processing, so that in read-mostly workloads, the compactness of the original key group index is maintained for optimal memory placement. Maintaining two indices slows down all write operations, so combining the two indices may be preferred if a more balanced read-write workload is expected.

For the operations described below, if one can avoid online insertions, updates, or deletions, then the modifications can be logged sequentially to disk. The operations can then run at a different time, or if enough operations are batched, the heapfile can be re-clustered for improved layout.

6.1 Insertions

When a tuple is inserted into a table, for each field on which a key group index exists, we must update all of the pages referenced by the complete key group index. We attempt to add the tuple to each page. If no room exists on the page, we delete the page slot entry and key group index entries which claim that this key group is fully contained on the page, guaranteeing the correctness of future queries. If no entry exists for the value in the complete key group index, either due to the aforementioned deletion or because a new key group is being created, we simply place the tuple at the end of the heapfile. Over time, this causes join path evaluation to traverse multiple pages to find all requisite key groups, necessitating re-clustering.

6.2 Deletions

A SQL DELETE statement utilizes a SELECT-like syntax to produce a set of tuples to be removed from a given table. We process the SELECT as described in Section 4.6, and take note of the tuples we must delete. For every tuple to be deleted, we then search the complete key group index for each key group that the tuple is a member of, and remove the tuple from the pages it is found on. If a deleted tuple is the last of its key group to be deleted, we remove the key group page slot entry for the pages it appears on, and remove the key group's entry from the appropriate indices.

6.3 Updates

An SQL UPDATE statement also uses a SELECT-like syntax described in the discussion on deletions. Because update performance is a secondary concern, one can achieve correctness simply by replacing the SELECT section of the UPDATE into a DELETE, noting which tuples were deleted. For all deleted tuples, one can then use the insert method described in Section 6.1. This avoids the complex accounting that would occur when several of a tuple's key group fields are updated to other key values.

Chapter 7

Experiments

We now evaluate the performance of BlendDB against a traditional relational database using a variety of materialization techniques. Our goal is not to show how BlendDB performs on all flavors of relational queries. We instead wish to demonstrate that a BlendDB-clustered database can perform better than a traditional RDBMS with standard normalization techniques using slightly more space. Next, we will show that BlendDB takes less space than a materialized view approach for commensurate performance. Finally, we seek to show that in exchange for increased space consumption, BlendDB provides improved performance over materialized views when a user navigates between related items of interest. We also explore some of BlendDB's weaknesses compared to traditional RDBMSs. For our performance tests, we focus on queries over the IMDB dataset [1], which we have placed into relational form. The schema and key/foreign key relationships of the IMDB dataset are shown in Figure 1-1. To summarize the dataset, there are 1.35 million actors, 1.06 million movies, 110 thousand directors, 8.39 million acted relationships, 690 thousand directed relationships, and 220 thousand movie ratings. The six files, one per table, with a newline separating tuples and one character separating each field, take 297 MBytes.

7.1 Implementation details

We implemented BlendDB in C++; our implementation follows the design in Section 4. Our implementation is consistent with all sections of this thesis, but does not include an implementation of the sequential scan indices described in Section 4.4, or the implementation of insertions, deletions, or updates described in Section 6. For B+Tree indexing, we use Berkeley DB 4.5 with 8 Kbyte pages. We also tried our experiments with Berkeley DB hash indices, but saw no noticeable difference. The clustering algorithm used is the sequential algorithm described in Section 5.1.

For comparison to a traditional RDBMS, we use PostgreSQL [5] version 8.3.0. We ran Postgres at the most relaxed isolation level and turned *fsync* off, although for the read-only queries that we ran, neither of these affect performance.

To ensure that performance differences between BlendDB and Postgres are not due to implementation details, our experiments establish that PostgreSQL is (slightly) faster than BlendDB when performing simple lookup queries for a single record from a table. We readily admit that our BlendDB implementation is not as feature-complete as PostgreSQL, but the comparison is valid because all operations we run are disk-bound, meaning that the lack of a SQL query processor and optimizer or locking found in traditional relational databases does not affect the comparison. To further justify our comparison, we run experiments with concurrent queries, and show that due to the disk-bound nature of the query processing, parallelizing resource consumption does not improve one database's throughput over the other, since queries which run in parallel end up depending on sequential disk access anyway.

Our test system is a 3.20 GHz Pentium IV with 2 Gbytes of memory, 1MB L2 cache, and a 200 Gbyte 7200 RPM hard drive. The system runs Linux (Fedora Core 7).

7.2 Database and Table Configurations

We now describe several database configurations that were used to compare various options for evaluating browsing-style queries. BlendDB was configured to cluster three join path sets in one clustering session, meaning that after the clustering is complete, BlendDB should be efficient at handling all three of the following join path sets:

1. Movie-oriented paths:

   {`movies.movieid`},

   {`acted.movieid`→`actors.id`},

   {`directed.movieid`→`directors.id`}, and

   {`ratings.movieid`},

2. Actor-oriented paths:

   {`actors.id`},

   {`acted.actorid`→`movies.id`}, and

3. Director-oriented paths:

   {`directors.id`},

   {`directed.directorid`→`movies.id`}.

We tested page sizes between 8 Kbytes and 1024 Kbytes in powers of two for BlendDB. This allows us to explore the benefits of co-locating more related items with each other on disk versus having to read more data at a time. Unfortunately, due to the way Postgres implements page slots for tuple offsets, one can only address up to a 32 Kbyte page. As such, the Postgres experiments ran on 8, 16, and 32 Kbyte pages.

As described in Section 5.3, another variable to explore in BlendDB is query path depth. To study the effect of this variable, we will compare two systems:

1. *BlendDB Path* 0: BlendDB clustered with a path depth of 0, meaning that in order for a path head object to be considered satisfied, only the tuples necessary to answer its own join path queries are required to appear on the disk page that it is on.

2. *BlendDB Path* 1: BlendDB clustered with a path depth of 1. In this case, a path head object is satisfied when the tuples of its own join path queries as well as those of related objects are found on the same disk page. For example, to satisfy a particular actor, we not only have to write both of his actor-oriented paths described above, but also have to write the four movie-oriented paths for each movie that he acted in. We hypothesize that this clustering improved throughput in situations in which a user views an object and then navigates to a related item, and show experiments to this effect.

For Postgres, we tested two physical schema configurations:

1. *Postgres Normal*: The basic schema in Figure 1-1 with a B+Tree index on all key and foreign key columns, physically clustering tables on the following indices: `actors.id`, `movies.id`, `directors.id`, `acted.movieid`, `directed.movieid`, and `ratings.movieid`. We chose to cluster `acted` and `directed` on `movieid` (instead of `actorid` or `directorid` respectively) because movies had twice the join paths for our join path set configuation in BlendDB, resulting in better performance on such joins. We report only movie-oriented query results when this schema is used, so the choice of clustered indexing on movie ID is optimal for these scenarios.

2. *Postgres MV*: In this approach, we create materialized views keyed on the path heads for each of the path types, one each for movies, actors, and directors. For example, we calculate the view for the actor-oriented paths by performing a LEFT OUTER JOIN between the `actors` table and the table generated by the query:

```
SELECT * FROM acted, movies
 WHERE acted.movieid = movies.id
```

The result of the LEFT OUTER JOIN is that any actor listed in the `actors` table will now be listed along with his movies, and in the degenerate case that he participated in no movies, his entry will still exist (these are the semantics of the LEFT OUTER JOIN, whereas the natural join would produce no results). This allows us to look up both the movies and the name of an actor with a single index lookup – that of the actor id. The expense is that the actor's information (the name and ID of the actor) is repeated as many times as there are movies for that actor.

To use materialized views, we manually rewrote the queries to utilize them. In general, automatically determining when a materialized view applies is a difficult problem that does not occur in BlendDB.

7.3 Clustering Evaluation

In this section, we show that the clustering algorithm described in Section 5 results in reasonably sized databases, and finishes in a reasonable amount of time.

Figure 7-1 shows the size of the heap files and indices used by various comparison systems, including the two PostgreSQL databases, on the IMDB database. For now, we only explore clustering in BlendDB to a path depth of 0, and so all references to *BlendDB* are with respect to *BlendDB Path 0.*

Note first that for the same page size (8 Kbytes), *Postgres Normal* takes about half the space of *BlendDB* (1 GB vs 2 GB). This can be attributed to the repetition of tuples by BlendDB. As we increase page size, we can reuse tuples more efficiently while clustering, as path heads that are discovered while writing are reused and written to the same page. Thus, *BlendDB* tends to take less space to store the database with larger page sizes. This trend is not enjoyed by either of the Postgres databases, because each of them stores a tuple instance once, and larger page sizes do not provide room for compaction.

Postgres MV's size (3.2 GB) is calculated by adding the three materialized tables for actors, movies, and directors in addition to the original six tables in Postgres. We

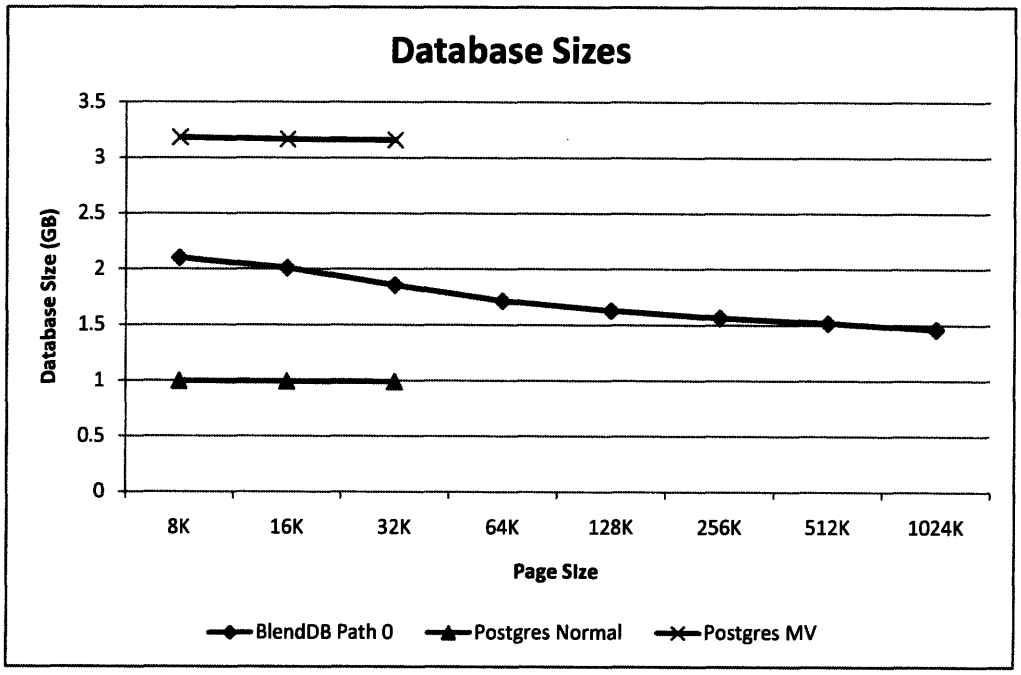


Figure 7-1: A comparison of the various database configurations in terms of on-disk sizes.

add the original six tables because one can not answer all queries from the materialized views alone. Note that this option is about 3 times the size of *Postgres Normal*, as repeated data in the materialized views can not be compressed out. Again, larger page sizes do not affect the overall disk usage, though in a database that supported compression of repeated data at the page level, we suspect that a similar trend to *BlendDB* could be seen.

Figure 7-2 compares the time it takes to build the various databases. Building and sorting (using the Postgres CLUSTER command) tables in *Postgres Normal* takes the least time for 8 Kbyte pages. Whereas *BlendDB* initially takes longer than *Postgres Normal*, it improves over *Postgres Normal* for larger page sizes, perhaps because less data is written to disk for larger page sizes in *BlendDB Path 0*. Because each level of joined tuples from the head of the first path written to a page grows with the join

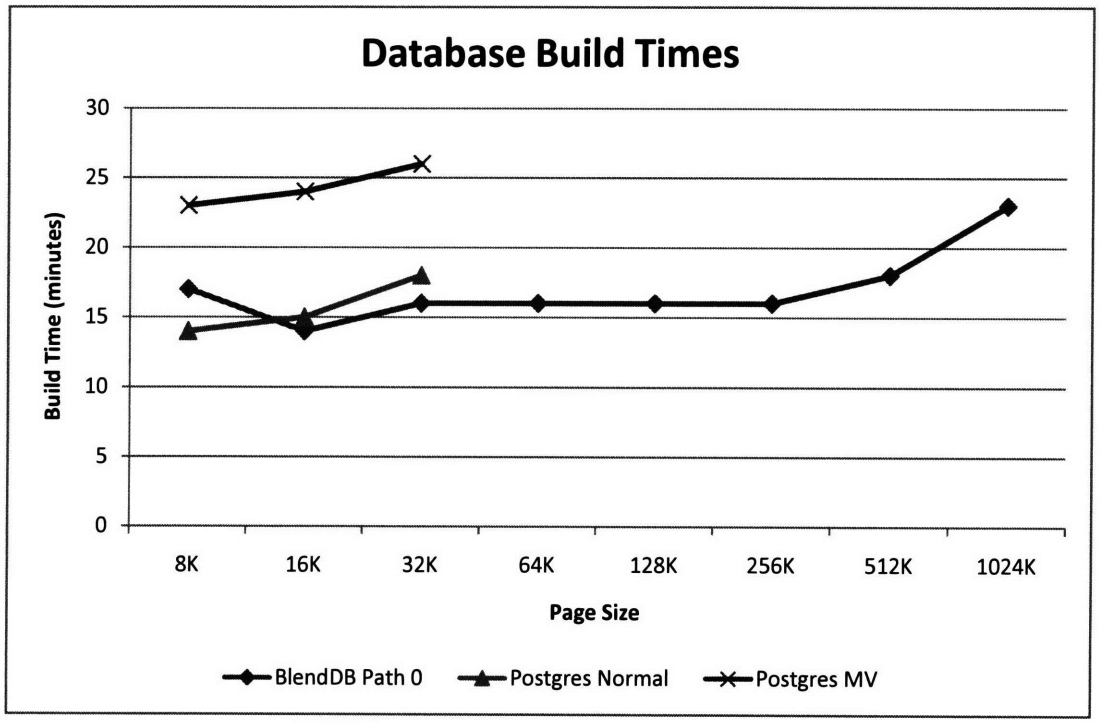


Figure 7-2: A comparison of the various database configurations in terms of the time to load, build, and cluster each.

fanout, larger pages require more resources to maintain the state of the clustering, thus affecting clustering speed of *BlendDB* for page sizes greater than 256 Kbytes. While *Postgres Normal* and *BlendDB* take comparable time for clustering, *Postgres MV* takes longer, as it has to cluster the three additional materialized views on top of *Postgres Normal.* Our goal is not to make a statement regarding the absolute speed of *BlendDB*'s clustering algorithm compared to the current state of the art, but simply to show that it is comparable to, and in some cases slightly faster than traditional RDBMS clustering techniques.

A more interesting experiment for the future is to select or construct datasets that are much larger than the available system RAM, and compare clustering algorithm performance on those datasets. Such an experiment would study the effects of disk seeks during clustering with traditional algorithms, as well as on the algorithm

described in Section 5.1 with the improvements discussed in Section 5.2.

7.4 Query Performance

We now move on to query performance. Since each path query runs relatively quickly, we measure the total time it takes to run a large number of queries, as described in each of the following experiments. This simulates the load on a database serving a large number of concurrent users, for example. Our experiments are all run in a single-threaded configuration with no-concurrency (we discuss experiments which verify that adding concurrency does not affect performance below). The experiments described are run three times per data point and averaged. Between each experiment instance, we drop the operating system disk cache and restart the appropriate database to clear its buffer pool.

We are primarily interested the disk performance of BlendDB compared to that of a traditional RDBMS represented by Postgres. The dataset we use has the potential to fit in memory over the course of many queries if we use large disk pages, and we must avoid this cache effect as it diverts attention and would not exist in a larger dataset. To accomplish this, all queries described are run on the test machine with the Linux kernel booted with boot argument "mem=512M," causing the system to use only 512 MB of physical RAM. We set the buffer pool size for table and index pages of both BlendDB and PostgreSQL to 30 MB, leaving the rest of memory for the operating system and OS disk cache. Of all of the page requests by the queries that ran against BlendDB, less than 5% were for pages that were accessed by an unrelated query. This means that cache effect did not significantly improve BlendDB's performance.

7.4.1 Single Index Lookups

The goal of the experiment shown in Figure 7-3 is to ensure that the comparison between Postgres and BlendDB is a fair one. Since queries in both systems are processed by using indices to make the initial lookups into tables and perform index-nested loops joins with related tables, our goal is to compare the throughput of basic

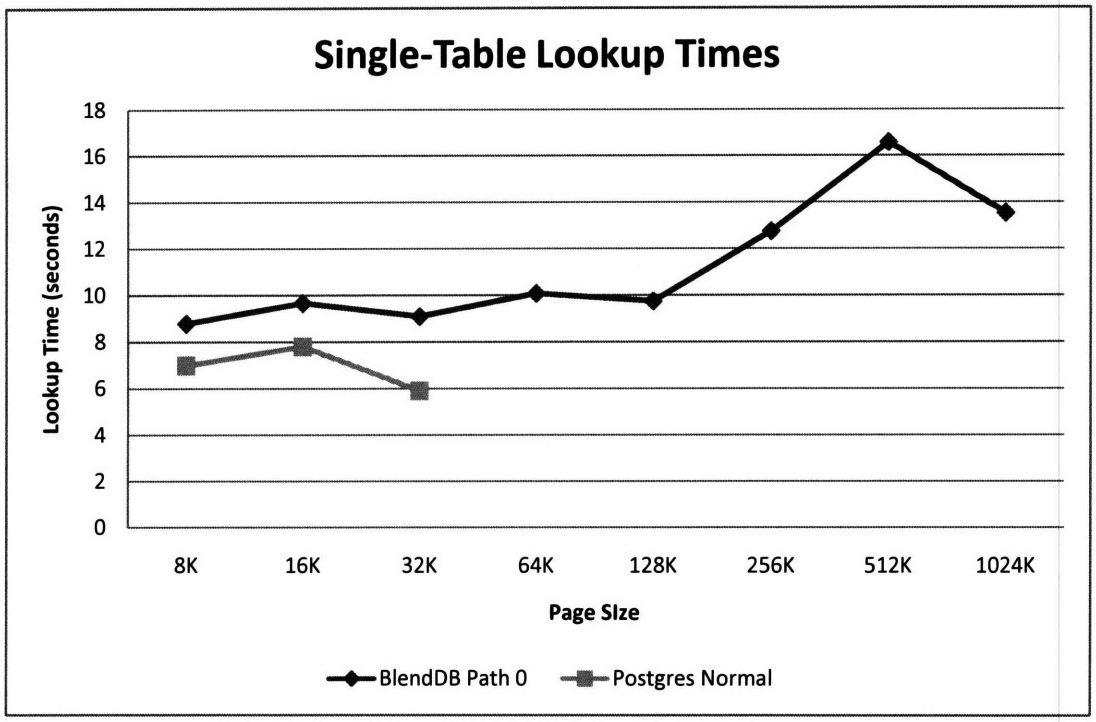


Figure 7-3: Postgres vs. BlendDB for simple index lookups

index lookups into a table in both systems. We test this by selecting 500 random movie IDs, and performing the query

`SELECT * FROM movies WHERE movies.id =` *ID*

for each ID. Because we only wish to compare the implementation performance of both systems, we only run this query on *Postgres Normal* and *BlendDB Path 0.*

Postgres fares better than BlendDB for all comparable page sizes. This is not surprising, as it is a robust, heavily optimized system, and its tables and tuples are smaller than BlendDB's, leaving more of the data in memory for future queries to access. Note that as page size increases, BlendDB's performance tends to stay the same or decrease, as sequential reads of large pages with only one desired tuple begin to dominate the cost initial random I/O. Since the index lookup peformance of Postgres is better than that of BlendDB, later experiments that show BlendDB outperforming Postgres using the basic building block of index lookups must be attributed to

reductions in the number of random I/Os afforded by BlendDB.

7.4.2 Concurrent Lookups

In order to further ensure that our throughput tests are fair, we also verified that all of our queries are disk-bound. All Postgres experiments showed less than 5% CPU utilization. The BlendDB experiments in Figure 7-3 also used less than 1% CPU, although in some of the following experiments BlendDB uses up to 10% of the CPU for larger page sizes due to in-memory sequential scans of entire pages. These scans occur when looking for key groups on pages that are brought into memory and reused later in a query to find co-located tuples, as discussed in Section 4.5.

Because queries are disk-bound and our system has a single disk, we don't expect concurrent queries to improve performance. To verify that this is the case, we ran the same experiments with two and four concurrent processes, each running one half or one fourth of the queries respectively. The results were unchanged in the case of Postgres and smaller page sizes in BlendDB, and and did not affect overall performance by more than 10% for larger disk sizes in BlendDB. Thus, we feel safe running all queries sequentially in both systems, and worry less about the additional query processing and optimization that Postgres is designed to perform, as these steps take insiginificant time compared to disk seeks for our workloads.

7.4.3 Movie Lookup Queries

In Figure 7-4, we show a performance comparison of running a procedure for each movie which queries for data required to display that movie's information in Postgres, BlendDB, and materialized views. For the materialized views scenario, all of a movie's data is available through a single lookup for the movie's id in the materialized view for the movies table. In Postgres and BlendDB, we run the following four queries for each movie id:

```
SELECT * FROM movies
 WHERE movies.id = ID
```

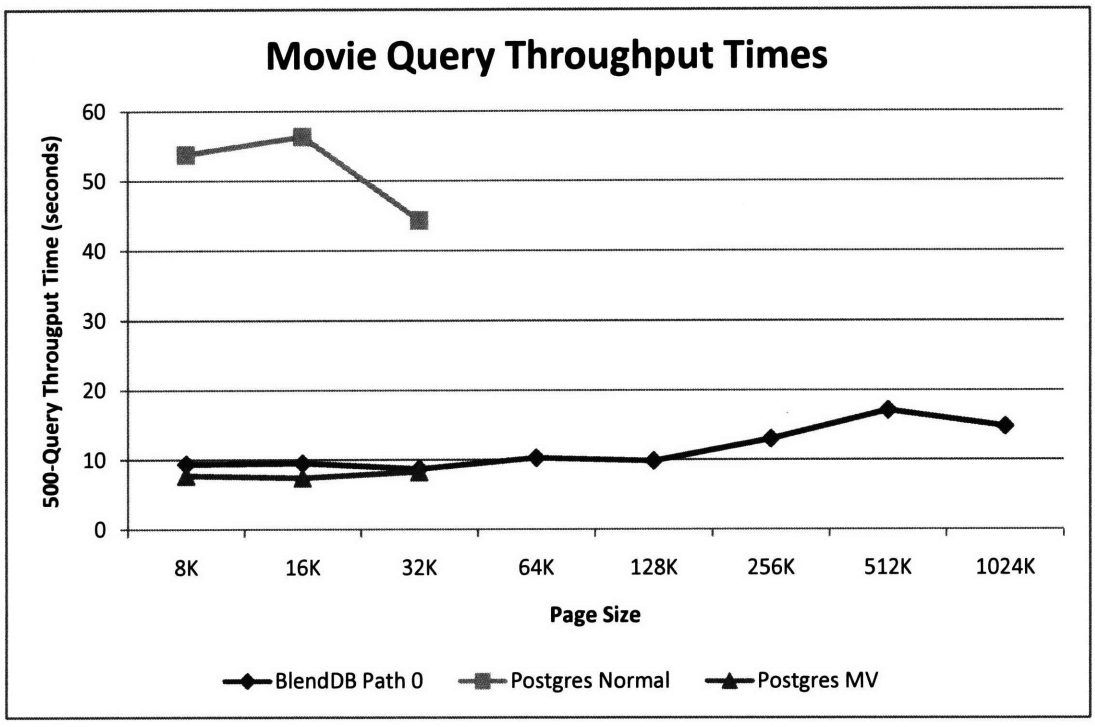


Figure 7-4: Comparison of PostgreSQL to BlendDB and materialized views by querying all 6 tables for 500 movies' data - actors, directors, movie names, ratings.

```
SELECT * FROM ratings
 WHERE ratings.movieid = ID

SELECT * FROM acted, actors
 WHERE acted.movieid = ID
 AND actors.id = acted.actorid

SELECT * FROM directed, directors
 WHERE directed.movieid = ID
 AND directors.id = directed.directorid
```

As before, the reported results are the aggregate time to look up information about all 500 movies.

BlendDB Path 0's most performant clustering (for 32 Kbyte pages) improves on *Postgres Normal*'s best throughput (also for 32 Kbyte pages) by a factor of 5.11, and the other page size clusterings have similar performance. This is the expected result—while Postgres performs at least one index lookup and heapfile lookup for

each of the four queries for a movie, *BlendDB* needs to perform a single index lookup and a single heapfile lookup to access all four queries for each movie, since all of the information for a movie is found on a single page for every movie in the 32 Kbyte clustering.

A second item of interest is the trend we see in *BlendDB*'s performance on this experiment and the index lookup experiment in Section 7.4.1. As page size increases, we see the same trend of decreasing performance after 32 Kbyte-sized pages, as all movies can be found on a single page, and there is no benefit to sequentially scanning more data. The absolute values of the timings of BlendDB in both experiments are also similar, even though the current movie lookup experiment answers four times as many queries as the index lookup experiment. Again, this occurs because the query results are co-located, so the disk utilization of the disk-bound queries does not significantly increase.

Postgres MV capitalizes on PostgreSQL's better index lookup facilities to achieve slightly better throughput than *BlendDB Path 0*. *Postgres MV*'s most performant clustering (16 Kbyte pages) performs 1.17 times faster than *BlendDB Path 0*'s best (32 Kbyte pages) in throughput. Unlike materialized views however, BlendDB does not require one to issue a modified query or select which table would answer the query most optimally - the database user simply issues the desired queries on the original schema. The commensurate performance of *BlendDB Path 0* also comes with reduced disk space utilization compared to *Postgres MV*, as shown in Figure 7-1.

7.4.4 Navigation Queries

We now show how BlendDB's clustering algorithm can trade disk utilization for performance when a user browses to items related to previously queried ones by changing path depth, as described in Section 5.3. So far, we have compared BlendDB to Postgres using a path depth of 0. While it is impractical for space reasons to increase the path depth past 1 (in the worst case, we see on-disk data inflating to 22.4 times the path 0 clustering) we show that the performance improvement of using path depth 1 can be significant.

Database Sizes and Build Times

	Database Size (GB)		Build Time(Minutes)	
Page Size	Path 0	Path 1	Path 0	Path 1
16k	2.01	8.95	14	48
32k	1.86	13.40	16	71
64k	1.72	19.58	16	108
128k	1.63	27.24	16	161
256k	1.56	34.95	16	229

Table 7.1: Comparison of *BlendDB Path 0* to *BlendDB Path 1* in terms of on-disk size of clustered databases and time to run clustering algorithm for various page sizes. The trend of decreasing disk space usage for *BlendDB Path 0* and increasing disk space usage for *BlendDB Path 1* as page size increases is discussed below.

Before considering the performance gains of path depth clustering, we first evaluate its disk space implications. Table 7.1 shows the on-disk utilization and clustering time for the IMDB dataset using a path depth of 0, as was done in Section 7.3, versus a path depth of 1. Note first that the disk space utilization of *BlendDB Path 1* is significantly higher than *BlendDB Path 0* (4.45 times as large for 16 Kbyte pages, and 22.4 times as large for 256 Kbyte pages). A similar trend is found with build times—larger page sizes require more space to cluster, and thus more time to compute and write out. Because the clustering algorithm for path depths of 1 does not consider a path head satisfied until its query result tuples and the query result tuples of all related path heads are written to the same page (when possible), fewer satisfied key groups can fit on the same page.

While the greater space requirements of *BlendDB Path 1* are to be expected, the greater space utilization required for larger page sizes merits attention. First, remember that in Section 7.3, we noticed a downward trend in space consumption with increasing page size for *BlendDB Path 0.* This comes as a result of repeated use of related tuples when selecting the next path head to cluster on a page. Once we move to *BlendDB Path 1*, however, more tuples are required to satisfy a single path head, and reuse is less frequent. The reversed upward trend in disk utilization with increased page size is due to a more subtle detail of the clustering algorithm.

In Section 5.4, we describe what we do with heads unsuccessfully written too

frequently to pages. Path heads that have been unsuccessfully written most frequently are selected from a priority queue, and if number of failed attempts passes a threshold (2), they are prevented from being written to future pages. This "adaptive" technique works best for smaller page sizes, as small pages are written more frequently, allowing for greater suppression of offending key groups. Because the path 1 clustering forces more tuples to a page than the path 0 clustering before a path head can be considered satisfied, more adaptive optimization is needed. For path 1 clustering, the adaptive technique is not effective enough with larger page sizes at keeping up with the path heads to suppress.

We provide some analysis to see where the clustering algorithm can benefit from heuristics to decrease disk space utilization. The first column of Table 7.2 ("Sorted by Key Group Size") shows what fraction of the heap file is occupied by the largest key groups for 16 and 256 KByte pages. The adaptive algorithm performs worse for 256 KByte pages than it does for 16 KByte pages, as can be seen by the greater percentage of the heap file that is occupied by the largest key groups. Since the size of a key group can be determined beforehand, one can predetermine which key groups to suppress instead of finding them adaptively, providing large savings. The second column of the table ("Sorted by Key Group Count") shows that the most frequent key groups also take a significant portion of the heapfile.

Furthermore, there is little overlap between the largest key groups and the ones that appear most frequently. The largest key groups (e.g., the large `Acted` table key group for an actor appearing in many movies) take a significant portion of the heap file, as do the most frequent key groups (e.g., the many instances of the small `Actor` table key group for each movie that references a popular actor). This suggests that if large key groups were suppressed *a priori*, and an adaptive algorithm suppressed key groups once they appeared too frequently, significant space savings could be achieved.

Further space savings could be achieved by using one of the statistical clustering methods described in the object-oriented databases literature [22], though we were unable to attain statisitical browsing traces for the IMDB dataset.

	Sorted by Key Group Size		Sorted by Key Group Count	
Percent of Top Key Groups	16k Path 1	256k Path 1	16k Path 1	256k Path 1
1%	10.2%	21.7%	15.3%	15.5%
5%	29.7%	38.2%	35.5%	36.7%
10%	36.6%	42.3%	47.4%	49.2%
20%	46.5%	53.4%	66.4%	75.7%

Table 7.2: Space consumption of the top key groups as a fraction of the total heap file size sorted by the size of the key group and sorted by the number of times the key group appears in the heap file.

Navigation Query Throughput

We now consider the throughput of queries in the case where a user browses from one item in the database to another related item. We explore the cases of a user beginning with a set of actors and exploring their related movies, and vice versa. For the actor queries, we select 499 random actor IDs, and issue two queries for each ID:

```
SELECT * FROM actors
 WHERE actor.id = ID

SELECT * FROM acted, movies
 WHERE acted.actorid = ID
 AND movies.id = acted.movieid
```

We then browse to between 0 and 8 related movies (whenever that many movies are available), and issue the same queries for those movies found in Section 7.4.3.

Figure 7-5 compares the best page size for in *Postgres MV* (16 Kbytes) and the best page size for *BlendDB Path 1* (128 Kbytes). Note that both *BlendDB Path 0* and *Postgres MV* take increasing time as more movies are navigated from the original set of 499 related actors. *BlendDB Path 0* takes less time than *Postgres MV* with more queries because even with path depth 0, the clustering algorithm is biased toward placing related items on the same disk page. The trend for *BlendDB Path 1*, however, is essentially flat—in almost all cases, the page for that actor also contains the tuples required to answer the queries for those movies. While both Postgres and the path 0 clustering for BlendDB initially perform better than the path 1 clustering, browsing for related objects allows for an up to 1.75x improvement. We see that for the (admittedly large) increase in disk space, the throughput of navigational queries

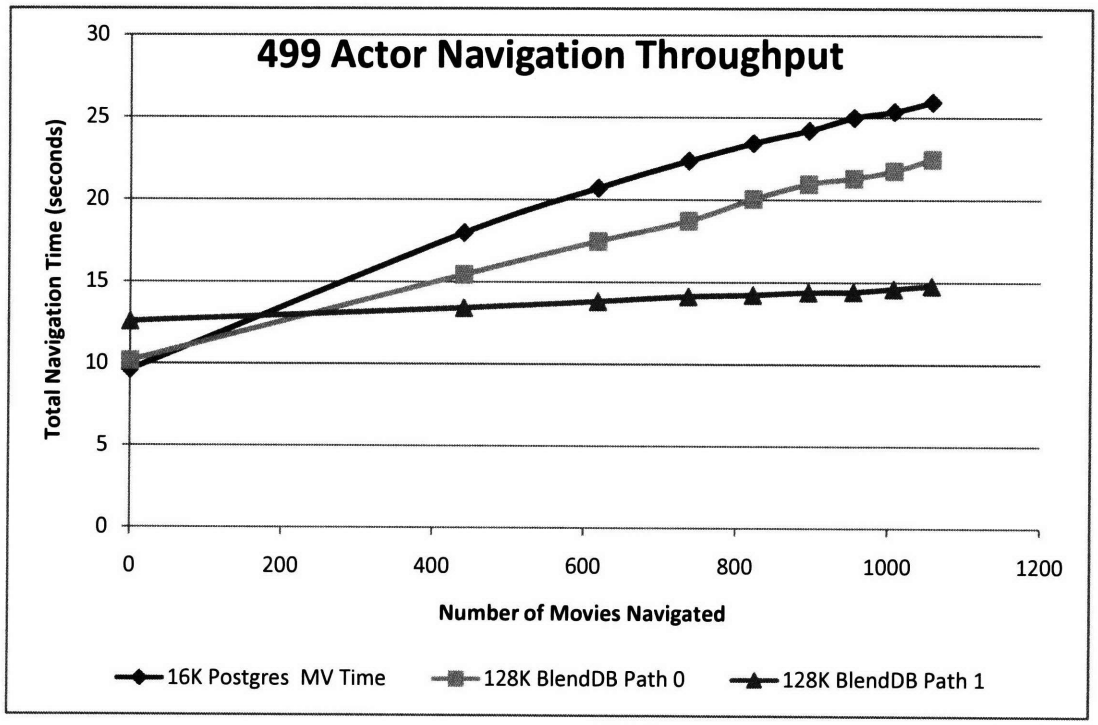


Figure 7-5: Comparison of the time required to navigate from 499 random actors to their related movies in BlendDB and Postgres MV. Measurements were taken at 0 through 8 related movies per actor, and counts along the x-axis represent the total number of movies navigated, since not all actors had that many movies.

can be substantially reduced.

To explore the reasons for improved throughput, Figure 7-6 compares the number of disk pages accessed when querying for the related movies using 16 Kbyte pages and 256 Kbyte pages. First note that the number of pages accessed when no movies are browsed (only the initial set of actors) is higher for 16 Kbyte pages, as some actors require more than one page to satisfy both actor queries. With larger page sizes, all actor queries are answered with a single page lookup, and so the number of pages accessed is on par with the number of actors requested. Furthermore, one can see that with path 1 clustering, both page sizes improve over path 0 clustering in terms of pages accessed. The 256 Kbyte path 1 clustering remains essentially flat with respect to browsed movies, indicating that all navigation queries are answered by the same

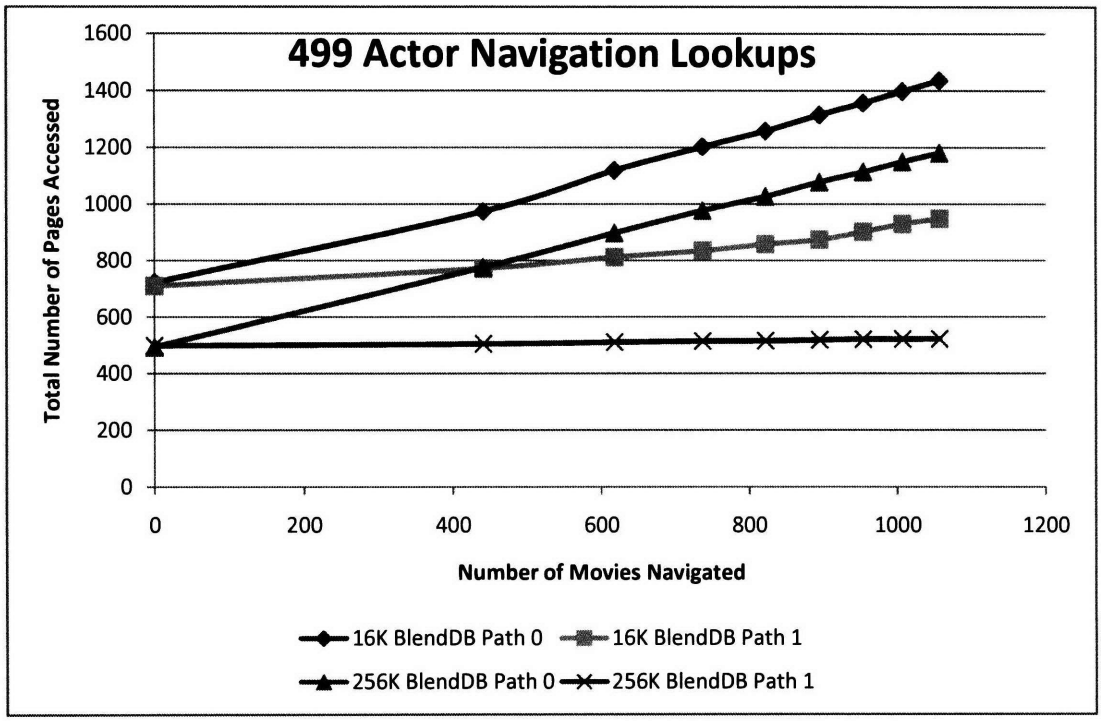


Figure 7-6: Comparison of the number of disk pages required to view movies related to 499 actors. Measurements were taken at 0 through 8 related movies per actor, and counts along the x-axis represent the total number of movies navigated, since not all actors had that many movies.

page.

We also include the results of navigation queries beginning at a set of 500 movies and navigating from 0 to 8 related actors in figures 7-7 and 7-8. The results for these experiments are similar to the ones above, with improved navigation and decreasing random I/Os in exchange for the increased space consumption of path 1 clustering.

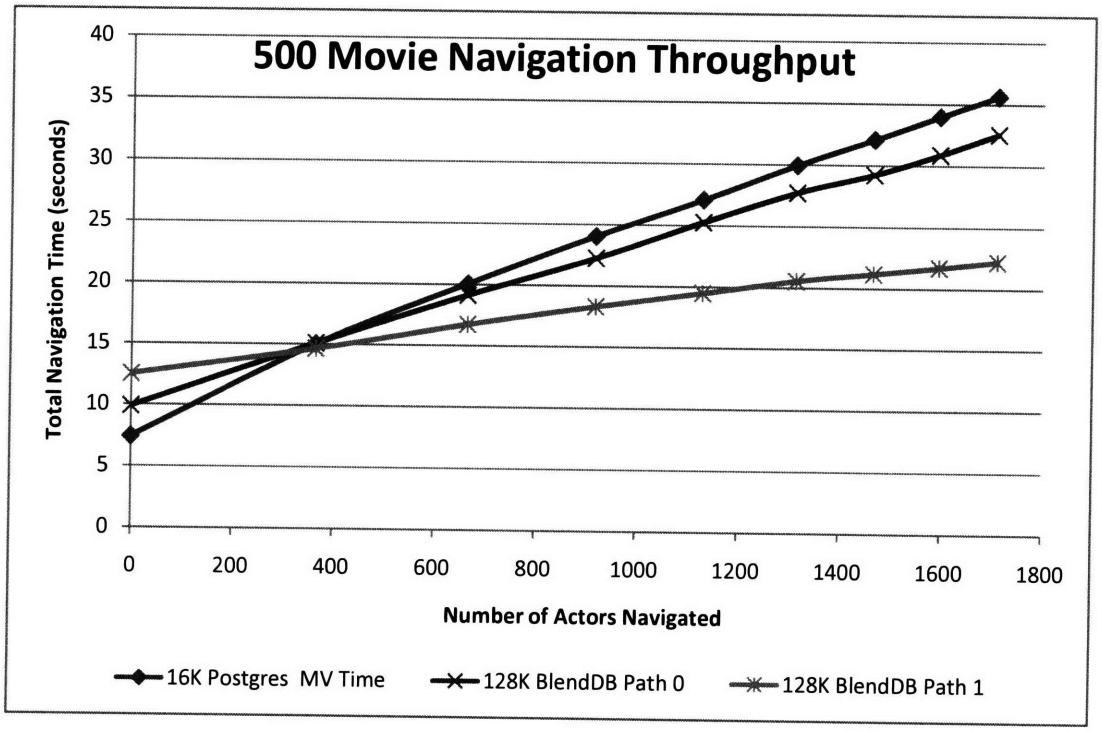


Figure 7-7: Comparison of the time required to navigate from 500 random movies to their related actors in BlendDB and Postgres MV. Measurements were taken at 0 through 8 related actors per movie, and counts along the x-axis represent the total number of actors navigated, since not all movies had that many actors.

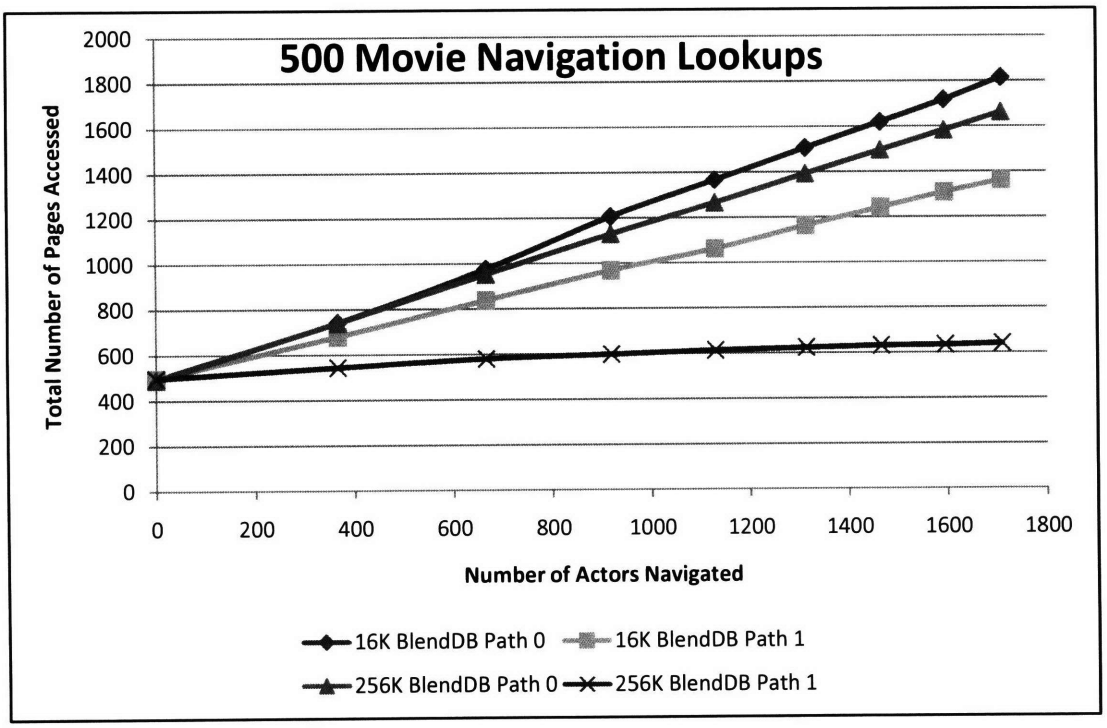


Figure 7-8: Comparison of the number of disk pages required to view actors related to 500 movies. Measurements were taken at 0 through 8 related actors per movie, and counts along the x-axis represent the total number of actors navigated, since not all movies had that many actors.

Chapter 8

Conclusions and Future Work

BlendDB was designed in the context of a modern browsing-style workload, in which relational databases are queried to construct objects that are described by multiple tables. Such queries naturally lead to a large number of inter-table references, but generally yield a small result set. Traditional RDBMSs following standard normalization practices fare poorly under such conditions, as they do not co-locate the data being accessed, which leads to manu random I/Os. Materialized views provide improved throughput in such query workloads, but require more space to denormalize schemas, create additional single-use tables, and lead to further complexity at query time when deciding which table should answer a given query.

BlendDB makes the same assumptions about a database designer's understanding of the access patterns to a set of tables as the materialized view approach does, but does not expose the physical reorganization of these tables on disk to the database user. At a baseline, it provides query throughput commensurate with materialized views while taking less disk space to store a database. Furthermore, it provides the semantics and facilities to improve throughput for navigational queries, where a user browses between related items, in exchange for more disk space to lay out the result of these browsing traces. Prior work in the OODBMS community on clustering object-oriented databases is complimentary to BlendDB's mission—if statistical information on the query workload and browsing patterns on a database is available, techniques originating in the OODBMS literature can be applied to BlendDB to improve nav-

igational throughput while reducing disk space utilization of infrequently-browsed data.

With this in mind, there are several directions for future work on BlendDB:

- The design of a system to instrument BlendDB at the server-side, or web browsers at the client-side, to study and report on statistical traces of user browsing behavior on certain datasets. Without this information, advanced clustering and space-utilization techniques can not be employed.

- The application of collected statistics and clustering techniques described by the OODBMS communty to BlendDB. While more of an engineering challenge, it would be interesting to study how such techniques differ in implementation between relational and object-oriented databases.

- Replacing the adaptive portion of the path depth clustering modification used by BlendDB with an algorithm that calculates *a priori* which key groups to suppress, allowing for better disk space utilization at larger page sizes.

- Implementing the scalable clustering algorithm of Section 5.2 to handle larger data sets, and studying the effects of this algorithm on growing data sets. Given this knowledge, one could explore a framework that allows programmers to implement various disk seek-oriented clustering techniques, which can then be scaled by using sequential scans in the framework.

- A study of BlendDB in a real-world high-throughput environment, with caching technology such as *memcached* reducing the random I/Os of the most popular queries.

- Further developing BlendDB's insertion, deletion, and update support, and studying the decreased performance of these implementations compared to traditional RDBMSs.

Bibliography

[1] Imdb dataset. `http://imdb.com/interfaces/`.

[2] Imdb website. `http://www.imdb.com/`.

[3] Memcached distributed object caching system. `http://www.danga.com/memcached/`.

[4] Oracle clusters description. `http://download.oracle.com/docs/cd/B14117_01/server.101/b10739/clustrs.htm`.

[5] Postgresql open source database. `http://www.postgresql.org/`.

[6] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System r: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[7] Veronique Benzaken, Claude Delobel, and Gilbert Harrus. Clustering strategies in o2: An overview. In *Building an Object-Oriented Database System, The Story of O2*, pages 385–410. 1992.

[8] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 190–200. IEEE Computer Society, 1995.

[9] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[10] Pamela Drew, Roger King, and Scott E. Hudson. The performance and utility of the cactis implementation algorithms. In *VLDB*, pages 135–147, 1990.

[11] Jonathan Goldstein and Per-Ake Larson. Optimizing queries using materialized views: a practical, scalable solution. *SIGMOD Rec.*, 30(2):331–342, 2001.

[12] Goetz Graefe. Master-detail clustering using merged indexes. *Inform., Forsch. Entwickl.*, 21(3-4):127–145, 2007.

[13] Bob Bryla Kevin Loney. *Oracle Database 10g DBA Handbook.* McGraw-Hill, 2005.

[14] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 95–104, San Jose, Calif., 1995.

[15] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajamaran. Indexing semistructured data, 1998.

[16] Patrick O'Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Rec.*, 24(3):8–11, 1995.

[17] Meikel Pöss and Dmitry Potapov. Data compression in oracle. In *VLDB*, pages 937–947, 2003.

[18] Johannes Gehrke Raghu Ramakrishnan. *Database Management Systems.* McGraw-Hill, 2003.

[19] Leonard D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.

[20] James W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Trans. Comput. Syst.*, 2(2):155–180, 1984.

[21] Manolis M. Tsangaris and Jeffrey F. Naughton. A stochastic approach for clustering in object bases. In *SIGMOD Conference*, pages 12–21, 1991.

[22] Manolis M. Tsangaris and Jeffrey F. Naughton. On the performance of object clustering techniques. pages 144–153, 1992.