ADAPTING DATABASE STORAGE FOR NEW HARDWARE

by

Allison L. Holloway

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2009
To my family.
ACKNOWLEDGMENTS

I would be remiss not to begin my acknowledgments by thanking my advisor David DeWitt. He has been a wonderful advisor to me, and I do not know where I would be without him – but I would most likely not be finishing my PhD. Our field is poorer due to his retirement from academia, but it is an honor to be able to say I am his last student. He once told me that my dissertation should be the best of his students’s. I am afraid that I could not live up to that request, but that is definitely not his fault. Well, except for the fact that he has picked some amazing students!

Next, I would like to acknowledge Jeff Naughton. His evenness, fairness and insights are definitely appreciated. I would also like to thank my other committee members: Jignesh Patel (one of David’s former students), Remzi Arpaci-Dusseau (and his last-minute substitute Michael Swift) and Mikko Lipasti. Further, I came to graduate school to study computer architecture, and I owe a debt of gratitude to Jim Smith, Guri Sohi, Mark Hill and David Wood, in addition to Mikko Lipasti, for convincing me to come to Wisconsin, and providing me support, advice and a solid academic foundation. Finally, I should thank Lizy John and Brian Evans from the University of Texas at Austin for encouraging me to go to graduate school and providing me good advice.

Shortly after I became David’s student, he sent out an email to his industrial contacts, to try to find a summer internship for a brand new database student with a strong computer architecture background. Mind you, this was late May/ beginning of June. Luckily for me, Laura Haas at IBM Almaden had a position available. Chapter 2 of this dissertation resulted from that summer internship, and Laura, Inderpal Narang (my boss) and Vijayshankar Raman and Garrett Swart (my mentors), deserve a lot of credit.

During my time in graduate school, I have been funded with two different fellowships: a National Science Foundation Graduate Research Fellowship and the NCR/ Teradata Anthony Klug...
Fellowship in Databases. I would like to thank the sponsors of these fellowships for helping to financially ease my way through graduate school.

A special note should go to my office mates: Jichuan Chang and Christine Reilly. Both have helped me through both my laughter and my tears – and there have been plenty of both. I would also like to give a shout-out to my qual study group of Mike Marty, Philip Wells and Matt Allen. We were four out of four in passing our quals on the first try, and will be four out of four in completing our PhDs, once Matt finishes later this year. Go us!

Kevin Lepak, who has been kind of like my architecture big brother, deserves thanks, and recognition for letting me know Wisconsin’s great academic programs. Jesse Davis should also be thanked for finally convincing me to switch from architecture to databases.

You cannot have spent as long in graduate school as I have without making some friends and meeting some fellow sufferers. I am horrible at categorization, so instead I will list these people, who have made my time here more enjoyable: Akanksha Baid, Jennifer Beckmann, Will Benton, Emily Blem, Mary Cain, Fei Chen, Eric Chu, Nick Coleman, Pedro DeRose, Alan Halverson, Natalie Enright Jerger, Kristen LeFevre, Christina Oberlin, Irene Ong, Alice Pawley, Ina Popovici, Eric Robinson, Warren Shen, Adam Smith and Sharon Whiteman. I must also thank my non-CS/Engineering ladies: Shaima Nasiri, Jean Merrill, Amihan Huesmann, Callie Langton and Jenna Weidner, and two long-time friends: Suzanne Rivoire and Catherine Griffin. Without these people, graduate school would seem even longer than it has been.

A special word of thanks should go to my Wisconsin relatives: my grandmother’s brothers and sisters and their children who have welcomed me into their family and homes. Judy Edwards deserves special mention for inviting me to family dinners for all the major holidays and going above and beyond in making me feel comfortable 1,000+ miles from home in the Great White (and did I mention COLD?) North.

Before I conclude, I must include Mrs. Parker, my high school computer science teacher. I took computer science my sophomore year to get the advanced credit and to satisfy my computer course requirement. She saw the talent in me that I did not see, or even think existed. Her three step method to various computer science and math fundamentals were definite lifesavers. (If only
she had one for getting your PhD!) While, without David, I would probably not be getting my PhD in computer science, without Mrs. Parker, I would not be in computer science at all. So: thank you, Mrs. Parker.

Finally, I have to thank my family: my parents Pat and Jay, my brothers Lane and Seth, my sister-in-law Andrea, and my grandparents Viola, Anne and Willie. Their love, faith and pride in me, and unflagging support (not to mention their stubbornness genes) have helped in more ways than I can count. Without them, I would not be who I am today, and I love them all.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LIST OF TABLES</strong></td>
<td>x</td>
</tr>
<tr>
<td><strong>LIST OF FIGURES</strong></td>
<td>xi</td>
</tr>
<tr>
<td><strong>ABSTRACT</strong></td>
<td>xiv</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Bandwidth and Compression Trade-offs in Database Scans</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Read-Optimized Databases, In Depth</td>
<td>3</td>
</tr>
<tr>
<td>1.3 DD: Trying to Solve Flash’s Random Write Problem</td>
<td>4</td>
</tr>
<tr>
<td>1.4 How to Extend the Buffer Pool with a Solid-State Disk</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Outline</td>
<td>5</td>
</tr>
<tr>
<td><strong>2 How to Barter Bits for Chronons: Compression and Bandwidth TradeOffs for Database Scans</strong></td>
<td>6</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Background and Related Work</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Data Representation</td>
<td>12</td>
</tr>
<tr>
<td>2.3.1 Column Coding</td>
<td>13</td>
</tr>
<tr>
<td>2.3.2 Tuple Coding</td>
<td>16</td>
</tr>
<tr>
<td>2.3.3 Block Coding</td>
<td>17</td>
</tr>
<tr>
<td>2.3.4 Fixed Length Quantized codes</td>
<td>20</td>
</tr>
<tr>
<td>2.4 Scan Generation</td>
<td>21</td>
</tr>
<tr>
<td>2.4.1 Short Circuiting</td>
<td>25</td>
</tr>
<tr>
<td>2.4.2 Parallelism</td>
<td>26</td>
</tr>
<tr>
<td>2.5 Experimental Evaluation</td>
<td>28</td>
</tr>
<tr>
<td>2.5.1 Methodology</td>
<td>28</td>
</tr>
<tr>
<td>2.5.2 Tokenization Performance</td>
<td>31</td>
</tr>
<tr>
<td>2.5.3 Predicate Evaluation Performance</td>
<td>33</td>
</tr>
<tr>
<td>2.5.4 Value of Parallelism</td>
<td>34</td>
</tr>
<tr>
<td>2.5.5 Discussion: Which Format is Better?</td>
<td>37</td>
</tr>
<tr>
<td>2.6 Conclusion and Future Work</td>
<td>39</td>
</tr>
</tbody>
</table>
# Read-Optimized Databases, In Depth

## 3.1 Introduction

## 3.2 Related Work

### 3.2.1 Database Compression Techniques

### 3.2.2 Vertical Storage

## 3.3 Factors Affecting Performance

### 3.3.1 Data Characteristics

### 3.3.2 Compression

### 3.3.3 Query Characteristics

### 3.3.4 Storage

## 3.4 Narrowing the Search Space and Generating the Tables

### 3.4.1 Table Parameters

### 3.4.2 Table Generation and Compression

### 3.4.3 Query Parameters

### 3.4.4 Scan Generation

## 3.5 Implementation Details

### 3.5.1 Choice of Code-Base

### 3.5.2 Read-Optimized Pages

### 3.5.3 Query Engine, Scanners and I/O

### 3.5.4 System and Experimental Setup

## 3.6 Results

### 3.6.1 Amount of compression

### 3.6.2 Effect of selectivity

### 3.6.3 Effect of Skewed Data Distribution

### 3.6.4 Wide Tables

### 3.6.5 Effect of Column Cardinality

### 3.6.6 Additional Predicates

### 3.6.7 Effect of “Wider” Attributes

### 3.6.8 Materialized Views

### 3.6.9 Joins

## 3.7 Discussion

## 4 DD: Trying to Solve Flash’s Random Write Problem

## 4.1 Introduction

## 4.2 Observations

## 4.3 DD: Dual Drives

### 4.3.1 Algorithm

### 4.3.2 Failure Recovery
## Appendix

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4 Experimental Setup</td>
<td>83</td>
</tr>
<tr>
<td>4.5 Results</td>
<td>84</td>
</tr>
<tr>
<td>4.5.1 Write-Intensive Workloads</td>
<td>84</td>
</tr>
<tr>
<td>4.5.2 Read-Intensive Workloads</td>
<td>85</td>
</tr>
<tr>
<td>4.5.3 Sensitivity</td>
<td>87</td>
</tr>
<tr>
<td>4.5.4 Comparison to NILFS</td>
<td>87</td>
</tr>
<tr>
<td>4.6 Related Work</td>
<td>88</td>
</tr>
<tr>
<td>4.7 Conclusions</td>
<td>90</td>
</tr>
<tr>
<td>5    Extending the Buffer Pool with a Solid State Disk</td>
<td>91</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>91</td>
</tr>
<tr>
<td>5.2 SSD-BM Implementation</td>
<td>93</td>
</tr>
<tr>
<td>5.2.1 Primary Buffer Pool</td>
<td>93</td>
</tr>
<tr>
<td>5.2.2 SSD-BM</td>
<td>95</td>
</tr>
<tr>
<td>5.2.3 Replacement Policy</td>
<td>99</td>
</tr>
<tr>
<td>5.2.4 Consistency</td>
<td>105</td>
</tr>
<tr>
<td>5.2.5 Recovery</td>
<td>105</td>
</tr>
<tr>
<td>5.3 Experiments</td>
<td>105</td>
</tr>
<tr>
<td>5.3.1 Disk Performance</td>
<td>106</td>
</tr>
<tr>
<td>5.3.2 System</td>
<td>106</td>
</tr>
<tr>
<td>5.3.3 Workload</td>
<td>106</td>
</tr>
<tr>
<td>5.4 Results</td>
<td>107</td>
</tr>
<tr>
<td>5.4.1 One Buffer Performance</td>
<td>108</td>
</tr>
<tr>
<td>5.4.2 Buffer Pool Reuse</td>
<td>109</td>
</tr>
<tr>
<td>5.4.3 Two Buffer Pool Replacement Policy</td>
<td>111</td>
</tr>
<tr>
<td>5.4.4 Multiple Update Streams</td>
<td>114</td>
</tr>
<tr>
<td>5.4.5 Skewed Data</td>
<td>114</td>
</tr>
<tr>
<td>5.5 Related Work</td>
<td>116</td>
</tr>
<tr>
<td>5.5.1 Flash and Hard Disk Drives, Together</td>
<td>116</td>
</tr>
<tr>
<td>5.5.2 Disk Drive Replacement</td>
<td>118</td>
</tr>
<tr>
<td>5.5.3 Buffer Pool Management</td>
<td>119</td>
</tr>
<tr>
<td>5.6 Future Work</td>
<td>119</td>
</tr>
<tr>
<td>5.7 Conclusions</td>
<td>120</td>
</tr>
<tr>
<td>6    Conclusion</td>
<td>121</td>
</tr>
<tr>
<td>6.1 Contributions</td>
<td>121</td>
</tr>
<tr>
<td>6.2 Applications and Future Directions</td>
<td>122</td>
</tr>
<tr>
<td>6.3 Closing Remarks</td>
<td>123</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table                                                                 Page

2.1 Schemas and Queries (LPR is l_extendedprice, LPK is partkey, QTY is o_quantity, OPR is o_totalprice, WK, DAYOFWK, YR represent o_orderdate, SNAT and CNAT represent s_nationkey and c_nationkey). ............................................ 29

2.2 Compressed sizes (bits/tuple) for various tuple encodings (columns) and field encodings (rows). ................................................................. 30

2.3 Size in bits of each column either uncompressed or with canonical Huffman coding applied. ................................................................. 31

2.4 Scan Speed (ns/tuple) for Q1a on Power-5 and Sun-T1 .................................................. 33

3.1 Type of compression used for each table. ................................................................. 51

3.2 Total tuple sizes with and without compression. ..................................................... 58
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Spectrum of compression formats from most to least compressed</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Generated code for scan over an 8-column table. Arrows denote instruction dependencies.</td>
<td>23</td>
</tr>
<tr>
<td>2.3</td>
<td>Scan Speed (ns/tuple) vs Compression Format as a function of the number of variable-length fields for queries that allow varying levels of short-circuiting (Q1a, Q1b, Q1c). Zero-B is for byte-aligned tuples, and zero-b is for bit-aligned tuples. The results are for a Sun-T1 with 32 threads and one interleave.</td>
<td>32</td>
</tr>
<tr>
<td>2.4</td>
<td>Scan Speed (ns/tuple) for Q2: one equality predicate on a fixed-length field and one on a variable-length field on the Sun-T1 with 32 threads and one interleave.</td>
<td>35</td>
</tr>
<tr>
<td>2.5</td>
<td>Scan Speed (ns/tuple) for Q3: 2 range predicates on the same variable length field on the Sun-T1 with 32 threads and one interleave.</td>
<td>35</td>
</tr>
<tr>
<td>2.6</td>
<td>Scan Bandwidth on the Sun-T1 for a scan (Q1b:top) and a predicate on a variable-length field (Q3:bottom)</td>
<td>36</td>
</tr>
<tr>
<td>2.7</td>
<td>CPU cost vs Memory/ I/O Bandwidth cost for chosen formats on the Sun-T1 with 32 threads and one interleave. Formats larger than 40 bits/tuple or that are dominated in both cycles/tuple and bits/tuple are omitted.</td>
<td>38</td>
</tr>
<tr>
<td>3.1</td>
<td>Query Engine Architecture [28].</td>
<td>56</td>
</tr>
<tr>
<td>3.2</td>
<td>Elapsed time for Narrow-E with various selectivity factors.</td>
<td>60</td>
</tr>
<tr>
<td>3.3</td>
<td>Elapsed time and CPU time for Narrow-E with 10% selectivity.</td>
<td>60</td>
</tr>
<tr>
<td>3.4</td>
<td>Elapsed time for Narrow-S at various selectivity factors.</td>
<td>62</td>
</tr>
<tr>
<td>3.5</td>
<td>Elapsed time for Wide-E at various selectivity factors.</td>
<td>63</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.6</td>
<td>Elapsed time for Narrow-U with various selectivity factors.</td>
<td>64</td>
</tr>
<tr>
<td>3.7</td>
<td>Three predicates on left-most columns (top) or random columns (bottom).</td>
<td>66</td>
</tr>
<tr>
<td>3.8</td>
<td>Elapsed time for String with various selectivity factors.</td>
<td>67</td>
</tr>
<tr>
<td>3.9</td>
<td>Materialized views for the Narrow-E table. The top figure presents the results for two materialized views with five columns in each view. The bottom figure presents the results for five materialized views with two columns each.</td>
<td>70</td>
</tr>
<tr>
<td>3.10</td>
<td>The top graph presents results for joining two two-column tables in a column store and row store. The x-axis labels give the storage paradigm and the selectivity of each scan. The bottom graph presents the results for joining two four-column tables.</td>
<td>72</td>
</tr>
<tr>
<td>4.1</td>
<td>Random and sequential read bandwidth (top) and write bandwidth (bottom) for a conventional disk (Conv) and a SSD (SSD).</td>
<td>79</td>
</tr>
<tr>
<td>4.2</td>
<td>Illustration of DD’s structures.</td>
<td>81</td>
</tr>
<tr>
<td>4.3</td>
<td>Write-intensive workload (80% writes).</td>
<td>86</td>
</tr>
<tr>
<td>4.4</td>
<td>Read-intensive workload (80% writes).</td>
<td>86</td>
</tr>
<tr>
<td>4.5</td>
<td>Sensitivity of the workloads to the buffer size for the write-intensive mix (case 1), with the MTRON drive.</td>
<td>89</td>
</tr>
<tr>
<td>4.6</td>
<td>Comparison of DD, the raw MTRON SSD and the MTRON SSD with NILFS, a log-structured file system, for the read-intensive mixes (cases 3 and 4).</td>
<td>89</td>
</tr>
<tr>
<td>5.1</td>
<td>Illustration of what happens in the SSD-BM when there is a miss in the primary buffer pool.</td>
<td>96</td>
</tr>
<tr>
<td>5.2</td>
<td>Illustration of what happens in the SSD-BM when a page is updated in the primary buffer pool.</td>
<td>97</td>
</tr>
<tr>
<td>5.3</td>
<td>Illustration of what happens in the SSD-BM when a page is evicted from the primary buffer pool.</td>
<td>98</td>
</tr>
<tr>
<td>5.4</td>
<td>Illustration of what happens in the SSD-BM when a page is flushed from the primary buffer pool.</td>
<td>99</td>
</tr>
<tr>
<td>5.5</td>
<td>Illustration of the One Buffer Replacement Policy.</td>
<td>102</td>
</tr>
</tbody>
</table>
Appendix

Figure | Page
--- | ---
5.6 Before (left) and after (right) page P8 is added to the SSD-BM in the Two Buffer Pool replacement policy. | 104
5.7 Geometric mean of the average elapsed times for each of the queries with different system configurations. The first bar is for the Intel SSD-only system, the middle bars are for different sizes (in GB) of the SSD-BM, and the last bar is for the HDD-only system. | 110
5.8 Same experiment as the previous figure, but with the MTRON SSD. | 110
5.9 Cumulative distribution function for the percentage of blocks that were used x times or less for One Buffer Pool SSD-BM configurations. | 111
5.10 Sensitivity analysis for an 8 GB SSD-BM with the Two Buffer Pool replacement policy on the Intel SSD. The percent tells the percent of the SSD-BM allocated to the cold region, and the > tells the threshold number of uses for promotion to the hot region. | 113
5.11 Geometric mean of average elapsed times for an SSD-BM of varying sizes using the Two Buffer Pool replacement policy on the Intel SSD. 25% of the SSD-BM is allocated to the cold region, and pages are promoted to the hot region if they have been used more than once while in the cold region. | 113
5.12 Geometric mean of average elapsed times for an SSD-BM of varying sizes using the Two Buffer Pool replacement policy on the MTRON SSD. 25% of the SSD-BM is allocated to the cold region, and pages are promoted to the hot region if they have been used more than once while in the cold region. | 115
5.13 Geometric mean of average elapsed times for the SSD-BM running a workload with one query thread and two update threads on the MTRON SSD. | 115
5.14 Geometric mean of average elapsed times for the queries over the skewed tables. The SSD-BM uses the One Buffer Pool replacement policy and the Intel SSD. | 117
5.15 Geometric mean of average elapsed times for the queries over the skewed tables. The SSD-BM uses the Two Buffer Pool replacement policy and the Intel SSD; the 8 GB SSD-BM configuration is also included for reference. | 117
ABSTRACT

Technology is always changing. In three years, today’s fast computer will be outdated, and, if not already obsolete, it will be fast on its way. This dissertation focuses on two hardware trends, processors getting faster and more powerful and the introduction of solid-state disks, and explores ways to take advantage of them in order to improve database performance. Processors have improved at a much faster rate than have memory and disk bandwidths. One way to overcome this difference is to use compression, which trades processing cycles for memory bandwidth to improve system performance. The first chapter explores the trade-offs in how much compression a system should use versus how much bandwidth the compression saves. The second chapter explores two different paradigms for storing data physically on disk, row stores and column stores, and how their performance compares for different queries and tables, when the data for both paradigms are compressed. The next chapter focuses on how to leverage a solid-state disk’s different read and write characteristics (compared to hard disk). We introduce the Solid-State Disk Buffer Manager (SSD-BM), a novel scheme where a solid-state disk is used as a secondary buffer pool. The pages can be accessed quickly thanks to the solid-state disk’s fast random reads, and they are only written sequentially, to overcome the solid-state disk’s sometimes poor random write performance.
Chapter 1

Introduction

Performance is a key issue in the design of a database management system (DBMS), and effectively using new technology is a crucial way to obtain high performance. For many years, this goal could be achieved by updating the processor and/or adding more memory to the system. However, database developers must now rethink key aspects of DBMS design to continue to take advantage of new hardware.

This change must occur because computers have changed drastically in the past two decades: processors have gotten faster and more complex, and hard disk capacity has risen exponentially. For instance, in 1991, a typical new computer may have been using a 25 MHz Intel 486 processor and a 40 MB hard disk. In contrast, a new computer today might have a 3 GHz, 4 core Intel Core i7 processor and a 750 GB hard disk. One can easily calculate that the hard disk has gotten 18,750 times larger, and, disregarding the “4 core” statement for now, the processor has gotten 120 times faster.

While the hard disk’s capacity increase is impressive, it does not tell the whole story, since if you care about performance – and if you are running a database management system (DBMS), you do – capacity is a lesser concern. Instead, designers focus on two important numbers: access latency and disk bandwidth. Access latency tells how long it takes from the time the request for data is sent to the time the data is found on disk and can start being returned to the processor. In 1991, the typical access time was 27 ms; in 2009, it was 8 ms – merely a factor of three improvement. Sustained disk bandwidth is how much data can be transferred from the hard disk per second. In 1991, disks provided a sustained bandwidth of around 0.7 MB/s; 18 years later, it is 70 MB/s. This
100-fold improvement is better than that for access times but is still less than the improvements seen in processors.

System designers aim to balance the component resources in order to get the best performance per dollar. This goal is very difficult to achieve when system resources have gotten so out of balance. If someone runs a workload with a lot of random reads and writes, where access times are incredibly important, their fast processor is essentially racing to a red light. And even if the workload only performs sequential scans of the data, the disk bandwidth has not improved enough compared to the processor to keep the processor’s instruction pipeline full. Additionally, most modern processors are now multi-core: in the case of the Intel Core i7, it is quad core - it has four processors on one die, all running at 3 GHz. So now the 70 MB/s bandwidth has to be split amongst four fast processors, which makes it even harder to keep the processors occupied.

Compression can help to re-equalize the imbalance between processors and disks. Compression trades processing cycles for improved effective disk bandwidth. The total bandwidth available remains the same, but more data can now be transmitted in the same amount of space. For instance, if the uncompressed table was 100 MB, and it compressed to 70 MB, the data could now be transferred in 1 second (1 s/ 70 MB * 70 MB) instead of 1.4 seconds (1 s/ 70 MB BW * 100 MB) – assuming decompressing the data does not consume too many processor cycles.

Thus far, we have only discussed devices that existed in both 1991 and 2009. But over the past several years, manufacturers have introduced a new type of device: the solid-state disk (SSD). SSDs have the same interface as hard disks, but have different underlying compositions and different performance characteristics. Solid-state disks are entirely digital and have no moving parts; they are constructed entirely using flash chips [10]. Because they have no moving parts, SSDs have much better random read bandwidth than hard disks. However, the flash chips that comprise an SSD are grouped together into roughly 256 KB blocks, and the blocks can only be written once; before the second write, they must be erased. This write-once characteristic leads to poor random write bandwidth because erasing take 1.5 ms. The SSD’s firmware, called the Flash-Translation Layer, tries to erase blocks in the background, but if there are no clean blocks, the erasure time cannot be hidden.
This dissertation studies techniques to adapt database storage to these hardware changes. The first two chapters focus on using compression and different storage paradigms to mask the differences in performance between hard disks and processors, while the last two chapters use solid-state disks in novel ways to improve system throughput.

1.1 Bandwidth and Compression Trade-offs in Database Scans

We first explore the trade-offs between compression and bandwidth for database scans. Compression techniques vary from the light-weight (such as dictionary encoding) to the heavy-weight (such as difference compression and Huffman coding). It takes very little extra processing to decode data that has been dictionary encoded, but Huffman coding, a variation of dictionary coding that results in variable length data, can take much longer. We present some performance optimizations for decoding Huffman-coded data and study how well different compression schemes compress a table.

While, in general, processors have improved at a much faster rate than disks have, different systems still have different ratios of processing cycles to available disk bandwidth. If tables are compressed using too heavy-weight a technique, the compression can actually result in worse overall performance, since it can move a system from being disk bandwidth-bound to being processing bound: from waiting on disk, to waiting on the processor. We quantify the costs of different compression schemes, and suggest ways to help determine how much compression should be used for a given system.

1.2 Read-Optimized Databases, In Depth

Traditionally, databases have been stored row-wise: tuples are stored together on disk. Recently, column stores have become a popular alternative storage paradigm. In a column store, the values in each column of a table are stored contiguously on disk. Column stores use less disk bandwidth than row stores, since only those columns used in a query must be retrieved from disk.
But they require the result tuple to be stitched back together, which incurs a processing cost. However, this trade-off can result in improved throughput and response times, particularly due to the growing difference in processor and disk speeds.

In Chapter 2, we find that row stores can be highly compressed, a claim many column store proponents downplay. Hence, we decided to study how well compressed row stores compare to compressed column stores for a given system with different tables and queries. We found that both column stores and row stores excel in certain situations, but the majority of the time, no paradigm dominates the other, since column stores can be significantly affected by processing costs.

1.3 DD: Trying to Solve Flash’s Random Write Problem

After studying compression and the growing difference between processor and disk speeds, we changed our focus to the emergence of solid-state disks. These devices have much better random read bandwidth than hard disks, but their random write bandwidth can be comparatively worse. Further, solid-state disks cost substantially more than an equivalently sized hard disk (for instance, a 256 GB SSD costs $579 from newegg.com, while the same sized hard disk costs $48). Because SSDs have the same interface as traditional disks, a system designer could simply replace hard disks with SSDs. However, because of price and performance differences between SSDs and hard disks, we wanted to study other, more novel, uses and system designs.

First, we noticed that the SSD’s write bandwidths were quite low, while the traditional disk had very good sequential write bandwidth, and it was must cheaper than the SSD. We drew on these observations and created a system that adds a hard disk to a solid-state disk to act as a write log. This design achieves much better write performance than the system with a solid-state disk alone, for only a small fraction more in price.

1.4 How to Extend the Buffer Pool with a Solid-State Disk

Finally, we use the same observations that guided our DD work to create the Solid-State Disk Buffer Manager (SSD-BM). The SSD-BM acts as a secondary buffer pool. It has fast random
reads, thanks to the SSD, and avoids random writes by managing the SSD-BM as a circular queue instead of using an LRU replacement policy. We study two novel replacement policies for the SSD-BM, and compare the performance of systems that store the DB entirely on disk, entirely on SSD, and using the SSD as a secondary buffer pool in various configurations.

1.5 Outline

The rest of this dissertation is organized as follows. Chapter 2 presents an analysis of the trade-offs made in compressing data, both in terms of the compression used and possible systems. Chapter 3 continues our work studying compression, but using two different storage paradigms for the data for one system - row stores and column stores. We then change our focus from processors and disks to solid-state disks. Chapter 4 presents a way to help mitigate solid-state drives’s poor random write performance by combining a SSD and a hard disk in one logical unit, and Chapter 5 presents a way to use a solid-state disk as an extension to the in-memory buffer pool. Chapter 6 concludes the dissertation.
Chapter 2

How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans

Two trends are converging to make the CPU cost of a table scan an increasingly important component of database performance. First, table scans are becoming a larger fraction of the query processing workload, and second, large memories and compression are making table scans CPU, rather than disk bandwidth, bound. Data warehouse systems have found that they can avoid the unpredictability of joins and indexing and achieve good performance by using massive parallel processing to perform scans over compressed vertical partitions of a denormalized schema.

This chapter presents a study of how to make such scans faster by the use of a scan code generator that produces code tuned to the database schema, the compression dictionaries, the queries being evaluated and the target CPU architecture. We investigate a variety of compression formats and propose two novel optimizations: tuple length quantization and a field length lookup table, for efficiently processing variable-length fields and tuples. We present a detailed experimental study of the performance of generated scans against these compression formats, and use this to explore the tradeoff between compression quality and scan speed.

We also introduce new strategies for removing instruction-level dependencies and increasing instruction-level parallelism, allowing for greater exploitation of multi-issue processors.

2.1 Introduction

In the past, scan speeds have been limited by the I/O data path and the layout of the data on external storage; thus improvements in scan performance have resulted from storage and file
system improvements. Recently, changes in technology have made table scans more interesting to database designers. This is because:

- Table scans give predictable performance over a wide range of queries.
- Disk space is now cheap enough to allow organizations to freely store multiple representations of the same data. In a warehouse environment, where data changes infrequently, multiple projections of the same table can be denormalized and stored. The presence of these materialized views makes table scans applicable to more types of queries.
- Scan sharing allows the cost of reading and decompressing a table to be shared between many table scans running over the same data, improving scan throughput.
- A table scan is embarrassingly parallelizable; the data can be spread over many disks and the scan can be performed by many processors, each performing its own portion of the scan independently. As hardware has gotten cheaper, it has become possible to tune the level of parallelism to control the response time.
- With the advent of parallel database appliances that automatically manage the individual processing elements inside the appliance, scale-out parallelism (e.g., clusters), which has long been much cheaper to purchase than scale-up parallelism (e.g., SMPs), has become competitive in terms of management overhead.
- Large addressable memories on inexpensive 64-bit processors allow entire databases to be partitioned across the main memories of a set of machines.
- Advances in database compression allow databases to be compressed to between 1/4 and 1/20 the original sizes in practice. This allows larger databases to fit into main memories, and it allows for a proportional effective increase in the data access bandwidth.

Additionally, compression is a key technology for table scans. One reason compression has generated so much interest recently is that, with the advent of multi-core processors, the rate at which arithmetic operations needed for decompression can be performed has been increasing more
quickly than the data access bandwidth rate [28]. This trend is expected to continue for many years and suggests that, eventually, we will be willing to pay for any amount of additional compression, even if it comes at the expense of additional arithmetic operations.

But, programs are not written to run “eventually:” they are written for and run on a particular target system; thus, the amount of compression must match the current system. In this chapter we explore how to generate code that optimizes the CPU speed at which data in a particular format can be scanned. Depending on the bandwidth and capacity of the data storage and the CPU speed, the format that best trades off compression and CPU utilization can be chosen.

There are presently three schools of thought on database compression:

- **Neo-compressionists:** This work focuses on substantially reducing the space used for representing data in the database, using techniques like run-length coding, Lempel-Ziv coding, entropy coding, delta codes, and extensive pre-compression analysis of the data (for example, [58, 52]). In particular, a recent paper [52] describes a method to compress a relation down to its entropy, and shows compression factors of 8 to 40 on projections of TPC-H data. These researchers do not use the standard slotted page format, instead they build compression blocks which have to be decompressed as a unit.

- **Strict vectorists:** These researchers match the compression format to the capabilities of the hardware to allow exploitation of multi-issue and vector capabilities. The data is stored column-wise, with each field, or small sequence of fields, encoded in a fixed-length machine data type [43, 15]. Domain coding is used extensively to reduce long and variable-length items to a small fixed size. This allows very fast queries and random access on the data, but the degree of compression achieved is not as good as when the entropy coding techniques are used. While this format can be updated in place because of its fixed-length nature, most implementations do not emphasize the speed of updates.

- **Slotty compressors:** Most commercial DBMSs have adopted a simple row-at-a-time compression format that builds on the traditional slotted page format but utilizes an off-page dictionary. These DBMSs support update in a straightforward way and have reduced the
engineering cost by decompressing pages when they are loaded from disk into the buffer pool \([31, 50]\). These formats are not optimized for scanning, but instead are designed to support update in place. We do not consider these formats further.

This chapter explores the spectrum of formats in between the maximum compression formats and the vectorized compression, while keeping the focus on scan speed. In our experiments we measure the CPU cost of the scan: the number of cycles consumed for the scan, select, project, aggregate, and group by, with the data stored in main memory. Our experiments start only after the data has been read into main memory. The effect of I/O on scans \([28]\) has been studied and data from a given I/O system could be folded into our CPU results.

Our scanning approach follows in the footsteps of MonetDB \([15]\): custom C code is generated for each query, rather than using interpreted code. This is needed to generate the best possible scan for a given implementation, and it is especially important when operating over compressed data because of the gains that can be made by customizing code, not just based on the data types and the query, but on the data encoding method and the actual contents of the dictionaries.

We start the body of this chapter (Section 2.3) with a detailed analysis of compression formats. We classify the variety of relation compression ideas that have been used in the past into a spectrum, ranging from highly compressed (and hard to scan) to lightly compressed (and easy to scan). Two of the recurring ideas in the compression of relations are difference coding and variable-length (Huffman) coding. We explore several relaxations of these two that trade off the degree of compression for scan speed. These relaxations form the spectrum of compression formats for our experimental study.

Prior researchers have made several assumptions about operating on compressed data, such as (a) bit alignment is too expensive and fields should be byte aligned \([8]\), and (b) variable-length fields are too expensive to decode \([8, 66]\). Two contributions of this chapter are:

- to experimentally verify these claims; our results (Section 2.5) suggest that the first is not valid, unless the scan is highly CPU bound.
two new optimizations, length-lookup tables and length quantization, for computing lengths of variable-length entities. Our experimental results suggest that with these two optimizations, Huffman coding does become viable for many queries.

Section 2.4 addresses scan generation. Superficially, a scan is simple, but the performance depends on a careful implementation of each detail. Accordingly, we discuss the ideas behind generating code to do a scan, and we identify several details that are vital for good scan performance: special handling of variable and fixed-length fields, techniques for short circuiting, and techniques to minimize conditional jumps.

We also discuss how to generate code that exploits the parallelism available on modern CPUs: code that exploits simultaneous multi-threading (SMT), multi-core and multi-issue instruction scheduling capabilities. We also present a new technique that increases the exploitable Instruction Level Parallelism (ILP) in a scan over a compressed table, which is called interleaving.

Finally, Section 2.5 presents our performance study. The experimental results are for two architectures, Sun Ultrasparc T1 and IBM Power 5, to illustrate the compression-speed trade off of various formats. Based on these results, we provide our main contribution: an outline of the rules and the data needed to determine the level of compression to use on systems with a given ratio of CPU speed to data bandwidth.

2.2 Background and Related Work

Work on database compression has been around for almost as long as databases themselves; even INGRES used null value suppression and some form of difference coding [59]. However, much of this research has tended to focus either on heavyweight compression with high compression ratios or lightweight compression with lower compression ratios, with the focus on both improved I/O system performance and decreased storage capacity.

Most of the early studies focused on compressing entire pages of the database. Though this led to good compression, page decompression was very expensive, particularly if only a few tuples were needed from the page. Graefe and Shapiro [25] were among the first to suggest tuple and
field-wise compression and operating on still-compressed data; they noted that many operations, such as joins, projections and some aggregates, could be performed on compressed tuples.

Dictionary based domain compression, a lightweight compression method where data values are mapped to fixed-length codes, is used in many database implementations and has been the subject of extensive research [13, 41, 50]. Entropy coding techniques, including Huffman encoding [30], are considered heavy-weight techniques, since decompressing variable length entropy encoded data is more processor intensive. Both Huffman and arithmetic encoding have been studied and modified to work better in databases [53, 20, 52, 65].

Early compression research mainly focused on disk space savings and I/O performance [23, 20, 31]. Later papers acknowledge that compression can also lead to better CPU performance, as long as decompression costs are low [53, 62, 47]. Some studies examine the effect of compression on main memory databases, and most look at choosing one compression scheme and finding the execution time savings from it [41]. However, [16, 8] find that query optimizers need to include the effects of compression in their cost calculations.

Commercial DBMS implementations have generally used either page or row level compression. IBM DB2 [31] and IMS [20] use a non-adaptive dictionary scheme. Oracle uses a dictionary of frequently used symbols to do page-level compression [50].

The MonetDB project has explored hardware compression schemes on column-wise relations [66] with the assumption that the tuples must be decompressed in order to query them, while we try to avoid decompression. Their research focuses both on compression and decompression performance while this chapter does not include the costs of compressing and tries to decompress as little as possible. [66] also presents a technique they call double-cursor, which has goals similar to what we call thread interleaving, increasing the available instruction level parallelism, but it is used to encode the tuples, and it does not physically partition the tuples.

C-Store [58] is a system that does column-wise storage and compression. It also delta codes the sort column of each table. Further, research has also been done to use compression in C-Store in order to decrease query times [8]. However, much of the focus is on light-weight compression schemes, such as run-length encoding, which are more suited to column stores than row stores.
This research supports our belief that it is best to decompress as little of the data as possible. C-Store researchers have also explored how expected query workloads should affect the column encoding and what the optimizer needs to know about the implications of direct operation on compressed data.

The careful study on scanning performance done by Harizopolous et al. [28] has shown the effect of compression on disk layout performance, showing how the cost of joining data from multiple disk streams can be compared with the cost of skipping over data from a single stream. Compression’s effect on the results was analyzed and, for many work loads, it changed the system from being I/O bound to being CPU bound. Their use of the cycles-per-byte system ratio to generalize the results beyond their test system and to understand the momentum of future systems has greatly influenced this research.

2.3 Data Representation

There are many different ways to represent data to allow efficient scanning; this section discusses a number of alternative formats and when they are useful. Multiple formats are examined because no single format is best in all cases: there is a tension between formats that allow for more efficient processing of the data and formats that represent the data more compactly. Depending on the relationship between where the data is stored and the access bandwidth to that data, and the processing power that is available for the data, one format or another will be more efficient. The consensus is that over time the ratio will tilt in favor of instructions, and this trend leads us to consider formats that are more heavily compressed and more expensive to process today.

This section begins by categorizing data formats on three dimensions, ordering the options within each dimension by decreasing processing speed and increasing representation efficiency. These dimensions are: column coding, the way that individual column values are coded to form column codes; tuple coding, the way that the column codes are combined to form a tuple code; and block coding, the way that multiple tuples are combined to form a compression block.

Individual database systems implement a particular point from each dimension that represents the trade offs made by the designers. For example, in the entropy-compressed format of [52], the
implementers have chosen the most highly compressed form in each dimension. The more lightly compressed format chosen by MonetDB [15] uses intermediate values in each dimension. Today, traditional databases, most of them designed when the compute-to-bandwidth ratio was skewed toward the bandwidth side, have focused on the easiest-to-process format.

We believe that, ultimately, the choice of data format will become an integral part of physical database design: this would involve analyzing the data set, computational environment, and workload to determine the best data format. For example, on a certain set of columns, variable-length entropy coding might save such a small amount that fixed-length domain encoding might be preferable, while on other, highly skewed columns, the reverse decision might be made.

Figure 2.1 illustrates the three dimensions of data formats, whose components are described below. Note that, in combination, this amounts to over one hundred formats, and the experimental section only gives results for those viewed to be most promising.

![Figure 2.1 Spectrum of compression formats from most to least compressed](image)

### 2.3.1 Column Coding

A column code is the string of bits that is used to represent a particular column value. We consider several ways of translating a column value into a column code.

*Machine coded* data represents each column value as a data type understood directly by the machine instruction set, that is, as tuples and arrays of fixed-length data items represented as one, two, four, or eight byte integers, characters, and floating point numbers. Values can be loaded into registers with a single instruction and some machines have special SIMD registers and instruction sets that allow operations on arrays of small values to be performed in one vector operation. Using
the machine data types minimizes the work needed to get the data into the registers of the machine in an efficient format. In addition, it is possible to randomly access into an array of such rows, which is important for following physical references to data items. This representation also entails the smallest amount of CPU overhead, since the data can be accessed with little more than a load and an array increment. However, this format is the most inefficient in space and, thus, bandwidth because it aligns column codes to machine boundaries, adds extra zero bits to the high end of words, and pads variable-length strings to their maximum lengths.

*Domain coding* assigns a fixed-length field to each column according to the cardinality of the column, not the value of the column. The mapping between the column code and the column value is maintained arithmetically, where applicable, or by using a dictionary and storing the index. The mapping used is stored in the meta-data of the representation. For example, if a CHAR(15) column can only contain seven distinct values, it can be coded into a 3 bit domain code. We can potentially improve the efficiency of accessing domain codes if the code length is extended to a byte boundary. For example, instead of coding the character string above with 3 bits, it could be padded to 8 bits (one byte).

*Huffman coding* assigns each column value a variable-length code chosen so that more frequent values within the column are assigned shorter codes. The important implication for scan efficiency is that fields are now variable length and their length has to be determined before the next field in the tuple can be accessed.

Canonical Huffman [52] coding is a reordering of the nodes of a Huffman tree so that shorter codes have smaller code values than longer codes and codes with the same length are ordered by their corresponding value. This format results in the same amount of compression as normal Huffman, but it supports both efficient code length computation as well as equality and range checking, without decoding or any access to a large data structure. We want to avoid accesses to large data structures since they may result in a cache miss, and cache misses are very expensive on modern processors.

Finding the length of a canonical Huffman code is important, but can be difficult. Once the length of the code is known, range queries can be efficiently performed directly on the code, the
code can be mapped into an index in a dense map of the values (for decode) or the same index can be used to access an array representing the results of an aggregation GROUPed BY the column.

There are two ways to compute the length of a canonical Huffman code. One option, presented in previous work [52], is to search in the Huffman mini-dictionary, which is a list of largest tuple codes whose first Huffman code has the given length. This can be done in time proportional to the log of the number of distinct code lengths. For example, if a canonical Huffman dictionary has codes: 0, 100, 101, 110, 1110, and 1111, the mini-dictionary will store the largest code for each code length left-shifted and padded with 1 bits to the word length of the machine, i.e. \{01111111, 11011111, 11111111\}. Thus, to find the length of the different fields in the tuple code 11101000, one would begin by comparing 11101000 to each of the three entries in the mini-dictionary, stopping when the mini-dictionary’s entry is no longer less than or equal to the tuple code, in this case at 11111111. The length associated with that entry is 4, so the first field is of length 4.

**Length-Lookup Table:** In this chapter, we introduce an alternative technique for computing the length using a *lookup table* indexed by a prefix of the tuple code. A Length Lookup Table replaces these multiple comparisons with a single array lookup using the code itself. The challenge is to keep this array small (ideally, to fit in the L1 cache).

A simple solution is to have an array LEN of $2^k$ elements, where $k$ is the longest code length. For the example dictionary, $k=4$, $LEN = \{1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 4, 4\}$. $LEN[0..7]$ are 1 because indexes 0..7 have a common prefix '0' in base 2. $LEN[8..13]$ are 3 because the indexes have prefixes 100/101/110. Now, the length of the code prefix in 11101... = $LEN[4$-bit prefix of 11101...$]

To tokenize the tuple code 11101000101: index into LEN with the 3-bit prefix 111, giving a code length of 4, and column code ’1110’; shift left by 4 to form 100101; index into LEN with
100 (or 4), so the length is 3, and column code is ’100’; shift left by 3 to get 0101; LEN[010] = 1, so the next column code is ’0’; shift left by 1, and LEN[101] = 3, so the last column code is ’101’.

The advantage of table lookup over binary search is that it involves no branches; mispredicted branches are quite expensive on modern architectures. The disadvantage is that the table can get long; table lookup is only efficient when the entire table fits in the L1 cache. So, as part of reading in a canonical Huffman dictionary, the mini-dictionary is always constructed, but the look up table is only constructed if it would be much smaller than the L1 cache.

Compression must take into account the processing cost, and dictionaries must be constructed to optimize efficient length computation and compression efficiency. The experiments presented in this chapter use the lookup table because of its better performance. To allow full exploitation of the length lookup table, a canonical Huffman dictionary generator was developed that matches a given histogram and has only one code length greater than 8. This limits the size of the length look up table to 256 bytes. The resulting code does not result in optimal compression, but for many column distributions it is quite good.

2.3.2 Tuple Coding

Once the column codes have been constructed, the next step is to combine the columns into a larger code that represents the entire tuple. There are two competing approaches for combining columns: a column store and the more traditional row store.

In a column store, each column is stored separately from the others; the correspondence between columns in a tuple is maintained either by using the same order of values between columns, or by augmenting each column value with a tuple identifier. The latter adds a substantial amount of extra size to the relation, and is not considered further. The former is essentially a column major layout of a table projection. Predicates are evaluated separately on each column, with bitmaps storing the results. The bitmaps are then anded together to form the results of the tuple predicate. Recent work comparing column stores and row stores [28] has shown that, for simple queries, storing the columns separately only adds a small amount of overhead, even when all columns are
required, and can save a great deal because of saved I/O when the query only requires a subset of the columns.

If the query workload is predictable, disk space is free, and a high update rate is not required, there is another alternative to either row stores or column stores: studying the workload and storing views of the more heavily used fields, and querying the projection that has the least number of extra columns, as is done in C-store [58]. This is the approach taken in this chapter. The results in this chapter use projections that are stored in row major order; we believe that our results should be applicable to either approach.

Because a tuple is formed by combining the column codes for each attribute, fixed-length fields are best placed at the front of the tuple. This allows the initial fields to be examined without waiting to determine the length of the previous variable-length column codes. There is some tension between this rule and the desire to order the columns to improve the efficacy of tuple difference coding. Tuples that are entirely comprised of fixed-length fields allow for greater parallelism within a scan, as the next tuple can be processed without waiting to determine the length of the current tuple.

When processing queries with low selectivity, there are many times when a predicate will fail based on an initial column code, which means the rest of the tuple does not need to be processed. In such a case, the evaluation of this tuple should be short circuited. But short circuiting requires knowledge of where the next tuple starts, which is non-trivial for tuples with many variable-length columns that must have their lengths individually decoded and summed. To simplify this process, we have added a variation on tuple coding where the total length of the tuple is explicitly and redundantly encoded as part of the tuple header, through a technique termed fixed-length quantized (FLQ) compression, which is described below.

### 2.3.3 Block Coding

Compression blocks are sequences of tuples that have been combined to form a unit so they can be updated and coded on a per block basis. Each compression block must be able to be interpreted independently in order to increase parallelism. Compression blocks are processed in parallel using
both thread-level and instruction-level parallelism. The compression blocks are sized so that thread processing overhead (e.g., pthreads) is dominated by the cost of scanning the block.

In our implementation, compression blocks each hold the same number of tuples. This makes it easier to scan several compression blocks simultaneously within a single thread, as will be described in the next section. Short compression blocks may be handled through a less optimized path.

The simplest and fastest approach to block coding is just to append the tuples codes. Since each tuple is either of fixed length, or the length can be determined by looking at the tuple, this is acceptable.

To increase compression between tuples, difference coding can also be applied. To perform difference coding, first the tuples are sorted lexicographically. The sorting and the differencing does not take into account any column boundaries. To maintain block independence, no differencing is done between blocks; the beginning of a block assumes a zero valued previous tuple.

The main idea of difference coding is that instead of storing the actual tuple codes, a compressed version of the difference between this tuple and the previous tuple is stored. Difference coding $m$ tuples saves approximately $\log_2 m$ bits [52]. For a terabyte of random data (a few billion tuples) the savings is 30-35 bits per tuple. In practice, difference coding correlated data can save even more bits per tuple.

There are two parameters that define difference coding: the difference operator and which previous tuple is used as a base for the differencing. The difference operators we support are prefix coding (also called XOR coding) and delta coding (which uses arithmetic subtraction). Difference coded tuples have the form: the first tuple, the number of consecutive zeroes at the head of the difference between the first and second tuples, then the tail (the numerical difference between the tuples), continued for all tuples in the block. For instance, the difference between 1023 and 1000 using prefix coding would be 1111111111 xor 111101000 = 0000010111. The number of zeroes at the head of the difference is 5, and the numerical difference between the tuples would be 10111. Thus, the two tuples difference coded would be 1111111111 0101 10111. The number of leading zeroes has been padded to 4 bits to include all the possible numbers of leading zeroes. We do not
have to explicitly store the first ’1’ at the head of the tail since, by definition, the tail begins at the first ’1’ in the difference. This optimization would lead to the code: 1111111111 0101 0111.

The number of zeroes at the head of the difference are normally encoded in order to save more space; these could be encoded using a Huffman code, but that slows down decode. To speed up the differencing decode, we use the same type of coding we use for the tuple length: a FLQ code, described below.

Our analysis and experience shows that delta codes are two bits shorter than prefix codes over random data. To see the advantage of delta coding over prefix coding, consider the coding of the numbers 1 to $m$. With delta coding the differences between codes is uniformly 1, which can be coded given the schema above in one bit: a zero length code and a one bit tail. With prefix coding, the expected length of the tail will be zero bits half of the time, one bit a quarter of the time, two bits one eighth of the time, and so on, which works out to 1 bit on average. Using a similar sum for the length of the code, the code will be two bits long on average, which results in a 3 bit code.

Even though prefix codes are a bit longer than delta codes, they are easier to process than delta coding for two reasons: multi-precision arithmetic is more complex than the multi-precision masking, and knowing the number of bits in common between two tuples allows us to reuse processing that was done on the first tuple. For example, if, by examining the first 57 bits of a tuple, we are able to determine that this tuple does not meet the selection predicate, then if the length of the identical prefix is 57 bits or longer, we can discard the new tuple without any additional processing.

However, any form of difference coding reduces the potential parallelism within the processing of a block, because the next tuple cannot be decoded until the previous tuple has been decoded. To mitigate this problem, reference coding has been proposed and implemented in MonetDB [15]. In reference coding, instead of representing the difference between pairs of tuples, we compute our differences from each tuple and the initial tuple in the block. In this way, after the first tuple has been computed, all the other differencing operations can go on in parallel. We generalize this notion and call the reference tuple the base of the differencing. Using a block level base instead of the adjacent tuple as a base can cost, on average, $\log_2 k$ bits if the block is of length $k$ (follows from Lemma 1 and 2 of [52]).
To split the difference between instruction level parallelism and compression, we propose using an intermediate base, we call the stride. A stride is a small sequence of tuple codes, typically 2, 4 or 8. Each tuple in the stride uses the same base, which is the last tuple code of the previous stride. Using this value costs slightly less than $\log_2 \text{stride}$. If the stride is chosen to match the level of loop unrolling or vectorization, then stride coding can be done with no extra overhead. Using a stride in this range allows for plenty of instruction level parallelism while not sacrificing nearly as much compression efficiency.

### 2.3.4 Fixed Length Quantized codes

Even with the advantages of canonical Huffman, variable-length columns still cost more than fixed-length columns to decode in terms of increased instruction count and reduced instruction level parallelism. For this reason we would like to use fixed-length codes to represent the tuple lengths and prefix lengths. These values have a large range, so how can we use a short fixed-length field to represent them? The answer is by quantizing the value, using what we call a Fixed Length Quantized code or FLQ code. Quantization works because we don’t actually have to represent the exact bit count for these fields: we are trading off compression for easier processing. We can always add more padding bits, as long as we know that is what they are. So, if we had a 45 bit prefix when all the rest of the prefixes were 48 bits, we could just add an extra three bits at the end.

As part of compression we run an optimizer that, based on a histogram of lengths, determines the optimal number of bits to use for the FLQ code and the optimal FLQ dictionary. An FLQ dictionary of a given length is an ordered list of values that are accessible by an index; thus, decode is just a fixed-length bit extraction of the index followed by a table lookup. The FLQ dictionary is computed using a recursive dynamic programming algorithm, taking into account the length of the FLQ code and the extra bits that have to be included due to quantization errors. Since the FLQ dictionary rounds the value represented up or down, we call it quantized compression.

For example, in our example TPC-H data set set the lengths of the tuple codes averaged 82.1, hitting every value between 81 and 110. Coding the length straight would have consumed 7 bits and using fixed-length domain code with arithmetic would have consumed 5 bits. Instead, using
an FLQ we coded the length in two bits, representing the quantized tuple code lengths 82, 83, 90, and 110. We then paid an average of 1 bit per tuple in quantization errors for a total cost of 3 bits per tuple. The cost for a Huffman code would have been 2.5 bits per tuple.

Similarly, the average length of the differential prefixes was 72.3 bits on average and ranged over 41 values between 8 and 90. This would require 6 bits as a domain code. FLQ used 3 bits to represent the prefix length, and paid an additional 0.7 bits per tuple for quantization errors which brings the total cost per tuple to 3.7, the same as the Huffman code cost.

2.4 Scan Generation

This section describe how to generate code to scan the tuples inside a compression block. Section 2.4.2 uses this skeleton as an atom to do a parallel scan over a table composed of multiple compression blocks.

At scan generation time, a lot of information is known about the data to be scanned and the query to be run. All of this information is hard-coded into the generated scan code:

- number of tuples in each compression block
- number of columns in the tuple
- the dictionary used to encode each column inside the tuple. This includes information about the relative probability of the values.
- the maximum size of the tuple code, computed by adding the maximum size of each column. This size is used to determine the number of 64-bit words needed to store a tuple, usually two or three. These words are declared in the scanner as separate automatic variables of type ‘unsigned long long.’ Having them as variables increases the chance that the compiler will keep them in a register.
• the query. We currently implement only a subset of SQL queries. Specifically, we consider only conjunctions of equality and range predicates on the columns and sum and count aggregates, with group-by. Queries always return at least one aggregate, since we wanted to focus on the CPU cost of the scan, and not the I/O cost of producing output tuples.

Figure 2.2 shows a skeleton of generated code for a scan over a table with 5 fixed and 3 variable-length fields. We will use this to explain the generation logic.

Our general scan model involves repeatedly fetching one tuple after another into local variables, and processing the tuple one constituent field at a time. The processing of fields within a tuple is hard coded (e.g., process field C and then F, G, and H) rather than loop-based – this is the advantage of a generated, as opposed to interpreted, scan.

2.4.0.1 Fetch Next Tuple

Within the main loop, the first operation is a call to nextTuple to fetch the next tuple in the compression block. nextTuple() is specialized to both the kind of block coding used in the compression block and the maximum length of the tuple\(^1\). If the block is coded via arithmetic deltas, nextTuple() reads the delta and adds it to the previous tuple. If it is coded via prefix deltas, nextTuple() shifts the delta and XORs it with the previous tuple. Both the addition and the XOR are multi-precision operations, but we have found that XOR is much more efficient than addition since there is no need to check for carries, which eliminates the potential for branch misses.

nextTuple() takes in a tuple offset and optionally returns the tuple length (if the format embeds the length). The tuple offset is set to 0 for the very first tuple in the block. For subsequent tuples, the offset is incremented in one of two ways:

• by summing the lengths of each of the fields as they are processed.
• by adding the embedded tuple length to the starting offset.

The latter is the basis for short-circuiting a tuple after processing only a subset of its fields (Section 2.4.1).

\(^1\)Even though the fields within a tuple are themselves compressed (via domain or Huffman coding), for most schemas the tuple takes up more than one machine word (64 bits).
FOR i = 1 TO NUM_TUPLES_PER_BLOCK DO

  /* Fetch tuple into local variables tRegs */
  nextTuple(tOffset, tRegs, &tupleLen, &sameBits);
  fldOffset = CONSTANT_OFFSET_OF_FLD_C;

  /* Operate on field C */
  fcode3 = getField(tRegs, fldOffset, &flen3);
  /* work on fcode3 */
  fldOffset = CONSTANT_OFFSET_OF_FLD_F;

  /* Operate on field F */
  vcode1 = getField(tRegs, fldOffset, &vlen1);
  /* work on vcode1 */
  fldOffset += vlen1;

  /* Skip field G */
  vcode2 = getField(tRegs, fldOffset, &vlen2);
  fldOffset += vlen2;

  /* Operate on field H */
  vcode3 = getField(tRegs, fldOffset, &vlen3);
  /* work on vcode3 */
  fldOffset += vlen3;

  /* go to next tuple */
  tOffset += fldOffset;

Figure 2.2 Generated code for scan over an 8-column table. Arrows denote instruction dependencies.
2.4.0.2 Processing Fields within a Tuple

nextTuple() is followed by a series of operations on fields within the fetched tuple. All the compressed formats considered place fixed-length codes first. Thus, all the fixed-length codes, plus the first variable-length codes are at constant offsets within the tuple. For these fields, the scan generator outputs a block of code for each field referenced in the query (C, F, H in Figure 2.2).

But the rest of the variable-length fields are at variable offsets, and have to be processed in order, even if they are not referenced in the query (like field G). Furthermore, the offset calculation means that operation on these fields is serialized, as shown by arrows in Figure 2.2.

Extracting a Field: The first step in operating on a field is to extract it from the tuple (still leaving it compressed). For a fixed-length field, getField() is just a shift and mask with constant. For a variable-length field, the mask depends on the length. getField() computes the length by either searching the mini-dictionary or through a table lookup, as discussed in Section 2.3.

A useful common-case optimization is for equality predicates. If the only operation on a field is an equality check with a literal, the mask is always a constant, even for a variable-length field, because it is determined by the length of the literal.

Operating on a Field: At this point, the field is Huffman or domain coded. If the query needs the field only for a predicate (equality or range), or for a group-by, coded field itself is operated on. For aggregations, the field is decoded.

Equality comparisons on Huffman coded fields are simple: the codes are checked for equality. Range comparisons exploit the order-preserving property of canonical Huffman codes, as suggested in [52]: at each level in a canonical Huffman tree, the values at the leaves are sorted in ascending order from left to right, thus, their Huffman codes are then also in ascending order. This means that if values $v_1$ and $v_2$ code to the same length, then $\text{code}(v_1) \leq \text{code}(v_2)$ iff $v_1 \leq v_2$. So, to calculate if $\text{field} > \text{literal}$, the scan generator pre-computes an array $\text{minCode}$ of the smallest code at each length (i.e., level of the Huffman tree) that satisfies the literal. Then the predicate is evaluated as $\text{code(field)} > \text{minCode[field.length]}$. A search of $\text{minCode}$ is not needed because the field’s length can easily be calculated from the length-lookup table.
Grouping and aggregation are done by maintaining an array of running aggregates, one for each group. In the absence of compression, group identification is typically done by a hash function. But group identification is easier for compressed fields, because the domain for canonical Huffman codes is dense within codes of each length.

For example, one might want to know the volume of sales grouped by the week of the year, to see which week had the most business. The aggregate statement would be \( \text{SUM}(\text{sales}) \) group by \( \text{week} \), and the resulting calculation would be:

\[
\text{sum}[\text{groupByIndex}(\text{week})] + = \text{decode}(\text{sales}).
\]

The \( \text{groupByIndex} \) maps the field code for week into a spot in the sum array. In this case, the mapping would be very simple, subtracting one from the week number (1-52). The field code for sales would also have to be decoded into its number value.

Canonical Huffman makes determining the index very easy since codes of the same length are consecutive integers. To compute the \( \text{groupByIndex} \), the length of the code is determined, as described in Section 2.3. The length of the code is then used as an index into a small array storing the precomputed count of the number of codes of shorter length. This value is then added to the index of this code within codes of the same length to get the result, e.g.,

\[
\text{groupByIndex}[\text{cf}.\text{length}][\text{cf}] \text{ and } \text{decode}[\text{ce}.\text{length}][\text{ce}],
\]

where \( \text{groupByIndex}[\text{len}] \) is an array of the aggregate-array indices for codes of length \( \text{len} \), and \( \text{decode}[\text{len}] \) is an array of the values for codes of length \( \text{len} \).

A domain coded field is treated the same as a Huffman code where the Huffman dictionary contains a single length. All the operations are exactly as for Huffman codes, but are implemented more efficiently because the length (and dependent quantities like \( \text{minCode}[\text{len}] \) and \( \text{groupByIndex}[\text{len}] \)) are constants within the generated code.

### 2.4.1 Short Circuiting

As discussed in Section 3, difference coded tuples may or may not embed the tuple length in the sequence of tuples. However, knowledge of the tuple length greatly impacts the scan. Without
knowing the tuple length, difference coding merely adds an extra step before the standard processing of the tuple, i.e. there will be an undifference tuple step after the nextTuple statement above, because the other fields’ lengths must be found in order to move to the next tuple. But if the tuple’s length is known, the next tuple can begin processing as soon as the current tuple fails the predicate, or the last aggregate is performed. The process of skipping to the next tuple as soon as possible is commonly referred to as short-circuiting.

The program flow in Figure 2.2 would include an “if” statement after each applied predicate to determine if the tuple needs to be processed further. Also, the tOffset increment would change to add the embedded length, rather than the fldOffset.

Short-circuiting increases the number of branches (since at least some predicates will fail), but the cost of mispredicted branches may be less than that of decoding the rest of the fields in the tuple. Also, short-circuiting difference coded tuples can allow us to re-use already computed information from the previous tuple, depending on how similar the previous and current tuples are, as was discussed in Section 3. This optimization comes at the cost of increased code complexity.

2.4.2 Parallelism

Today’s processors are highly parallel: most new and upcoming processors have multiple threads (most often of a type called SMT) and/or multiple cores on one die (CMP). While processors and compilers are good at extracting instruction level parallelism, they cannot split a stream of data into multiple parallel streams of data (partitions); it is something we, as programmers, must do for it.

The tuples are partitioned using one of two methods: evenly spaced cursors into the data or compression blocks.

If the tuples are only Huffman coded, the tuples can simply be partitioned by providing different pointers into the file. However, if the tuples are difference coded, compression blocks must be formed, since one tuple is dependent on the previous tuple, thus starting in the middle would render the compression unparseable. With compression blocks, difference coding must periodically be
started anew. Each new start of difference coding signals the start of a new compression block, and all but the last compression blocks are forced to have the same number of tuples.

2.4.2.1 Interleaving

Breaking the tuples into independent blocks allows us to simultaneously perform the same actions on each block of tuples, all in one thread. This parallel action on independent blocks within one processing core, or interleaving, allows us to exploit the superscalar properties of processors: they can execute multiple instructions per cycle, as long as they have instructions to execute. If only one block of data is being processed, dependencies within the data (for instance, a control dependence from a branch, or a data dependence on the length) limit ILP. But, if there are multiple blocks to work on - say 1-3 more, the processor can have more instructions to execute, thus reducing stall time. The example scan in Figure 2.2 only has one interleave. To modify it to include multiple interleaves, the same code block would be repeated multiple times within the loop, each working on a different sequence of tuples.

2.4.2.2 Threading

Today’s code needs to be threaded in order to fully exploit SMTs and CMPs. We use pthreads and multiple compression blocks to achieve this. The scan code itself might also include interleaving, in which case more pointers into the sequence of tuples or compression blocks will need to be passed to the scan. Each thread consists of the code loop illustrated in Figure 2.2, and the necessary pthread and data structure initialization code.

2.4.2.3 Orthogonality

The different forms of parallelism we introduce are, in general, orthogonal to other issues. However, different configurations may have different optimal amounts of exposed parallelism, since some possibilities are so compute-intensive that no additional parallelism needs to be provided. The next section further discusses these issues.
2.5 Experimental Evaluation

This section presents an extensive performance evaluation of our generated scans. It also discusses how much of the processors’ parallelism was able to be exploited and the effectiveness of short circuiting.

Different formats have widely varying compression ratios and hence (memory/disk) bandwidth requirements: formats that need less CPU to operate over the data tend to need more bandwidth to scan the data. Any complete system that uses one of these formats would have to have an I/O and memory subsystem that can support this bandwidth: a format that compresses poorly will need a powerful memory/disk subsystem, while a format that scans poorly will need a powerful processor.

The focus of this chapter is not to present a single such balanced system, but is instead to lay out the trade off between compression (bandwidth requirement) and scan speed (CPU requirement). The bandwidth requirement is directly calculated from the compressed tuple sizes (bits/tuple). We measure the CPU requirement by measuring the scan speeds (ns/tuple) on two different architectures. To make sure that our scans are entirely CPU-bound, the data sets were kept entirely memory resident.

Section 2.5.2 studies the performance of scans without predicates to measure the tokenization efficacy of each format. Section 2.5.3 studies scans with predicates. We investigate how well our queries can be sped up using processor parallelism in Section 2.5.4. Finally, Section 2.5.5 discusses these results in terms of the number of instructions/byte that each format needs, and correspondingly which format is appropriate for which architecture.

2.5.1 Methodology

The experiments were performed on 2 machines: a 2-core Power 5 (running AIX) and an 8-core 1GHz Sun T2000 server (running Solaris 10). The relative performance of each compression format is almost the same on the two machines, so most results are reported for the Sun only.

We use a TPC-H projection with attributes from the natural join of the Lineitem, Customer, and Supplier tables (Table 2.1). As in [52], two kinds of skew are added: (a) the customer and
supplier nations distribution are chosen per the distribution of Canadian trade from the W.T.O, and (b) the dates distribution ranges until 10000 A.D., with 99% in 1995-2005, 99% of those being weekdays, and 40% of those are in the two weeks around Christmas and Mother’s day. All other columns used the standard (uniform) TPC-H data distribution. Dates are stored as year, week, and day-of-week because there is often skew on weeks and day-of-week (e.g., more sales on weekends and before Christmas). Thus, the schema consists of 5 fixed-length fields (LPK through WK), and up to 4 variable-length fields (DAYOFWK through CNAT) that can benefit from Huffman coding.

Table 2.1 Schemas and Queries (LPR is _extendedprice, LPK is partkey, QTY is _quantity, OPR is _totalprice, WK, DAYOFWK, YR represent _orderdate, SNAT and CNAT represent s_nationkey and c_nationkey).

<table>
<thead>
<tr>
<th>Schema</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1a</td>
<td>select sum(LPR) CNAT from table group by CNAT</td>
</tr>
<tr>
<td>Q1b</td>
<td>select sum(LPR), DAYOFWK from table group by DAYOFWK</td>
</tr>
<tr>
<td>Q1c</td>
<td>select sum(LPR), QTY from table group by QTY</td>
</tr>
<tr>
<td>Q2</td>
<td>select sum(LPR), QTY from table group by CNAT where LPK=1 and CNAT=USA</td>
</tr>
<tr>
<td>Q3</td>
<td>select sum(LPR), QTY from table group by CNAT where LYR&gt;1990 and LYR&lt;8515</td>
</tr>
</tbody>
</table>

By default, the TPC-H data generator produces two kinds of correlation in this schema: LPR is functionally determined by the LPK, and the OPR is LPR × QTY.

The data set models a size of 1B rows, but, to keep the experiments manageable, only a 32M row slice of it was generated in sorted order; queries were only run on this slice. The experiments were explicitly designed so that I/O bandwidth would not be an issue, since we wanted to measure the CPU cost of operating over each format and the bandwidth needed for each format.

We study the speed of scans over different compression formats created by 4 tuple encodings and 3 field encodings:

- The tuple encodings arise from whether the tuples in the block are difference coded (D) or just appended (A), and whether or not lengths are embedded (via the fixed-length quantization introduced in Section 2.3) (L): they are denoted as delta (D), append (A), delta+len (DL), append+len (AL) in the plots.
- The field encodings are: all fixed-length byte aligned, all fixed-length bit aligned, and Huffman coded (with the length-lookup table optimization). The schema has 4 fields with skewed distributions: DAYOFWK, YR, SNAT, CNAT. So, the study has encodings with 1, 2, 3, and 4 Huffman coded fields. The resultant six field encodings are marked as zero-B for byte alignment, zero-b (for bit alignment), 1V (bit), 2V (bit), 3V (bit), 4V (bit) in the plots to indicate the number of variable-length fields the scan operates on.

These formats and the compression sizes they obtain are tabulated in Table 2.2. Results for each column are presented in Table 2.3. The uncompressed size is 320 bits/tuple. In contrast, the most compressed format considered, 4V.D, takes up 16.68 bits/tuple, and the entropy (calculated from the distribution) is 11.86 bits per tuple. The 4.82 bits/tuple difference is due to (a) not co-coding the correlated fields, (b) XOR coding differences rather than Huffman coding differences, and (c) using Huffman coding instead of arithmetic coding.

Table 2.2 Compressed sizes (bits/tuple) for various tuple encodings (columns) and field encodings (rows).

<table>
<thead>
<tr>
<th>Field Encodings</th>
<th>Length Embedded (AL)</th>
<th>Vanilla (A)</th>
<th>Delta Coded + Length Embedded (DL)</th>
<th>Delta Coded (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0V (byte): Byte-aligned all fixed</td>
<td>N.A</td>
<td>128</td>
<td>43.04</td>
<td>43.04</td>
</tr>
<tr>
<td>0V (bit): Bit-aligned all fixed</td>
<td>N.A</td>
<td>105</td>
<td>35.56</td>
<td>35.56</td>
</tr>
<tr>
<td>1V (bit): Bit-aligned 1 var len</td>
<td>101.66</td>
<td>100.66</td>
<td>32.25</td>
<td>31.24</td>
</tr>
<tr>
<td>2V (bit): Bit-aligned, 2 var len</td>
<td>98.32</td>
<td>96.32</td>
<td>28.91</td>
<td>26.91</td>
</tr>
<tr>
<td>3V (bit): Bit-aligned, 3 var len</td>
<td>87.46</td>
<td>85.43</td>
<td>19.48</td>
<td>17.46</td>
</tr>
<tr>
<td>4V (bit): Bit-aligned, 4 var len</td>
<td>86.81</td>
<td>84.24</td>
<td>19.25</td>
<td>16.68</td>
</tr>
</tbody>
</table>
Table 2.3 Size in bits of each column either uncompressed or with canonical Huffman coding applied.

<table>
<thead>
<tr>
<th>Column</th>
<th>Uncompressed</th>
<th>Can. Huff</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPK</td>
<td>20</td>
<td>N.A.</td>
</tr>
<tr>
<td>OPR</td>
<td>18</td>
<td>N.A.</td>
</tr>
<tr>
<td>QTY</td>
<td>6</td>
<td>N.A.</td>
</tr>
<tr>
<td>LPR</td>
<td>24</td>
<td>N.A.</td>
</tr>
<tr>
<td>WK</td>
<td>6</td>
<td>N.A.</td>
</tr>
<tr>
<td>DAYOFWK</td>
<td>4</td>
<td>2.81</td>
</tr>
<tr>
<td>YR</td>
<td>15</td>
<td>4.11</td>
</tr>
<tr>
<td>SNAT</td>
<td>6</td>
<td>1.66</td>
</tr>
<tr>
<td>CNAT</td>
<td>6</td>
<td>1.66</td>
</tr>
</tbody>
</table>

2.5.2 Tokenization Performance

Query 1 examines the cost of tokenizing a compressed table into tuples and fields. This is measured by running queries Q1a, Q1b, and Q1c: aggregation and group by queries that have no predicates. The three queries differ in the extent of short-circuiting they permit over the variable-length fields. Q1a, a grouping on the very last field (CNAT) allows no short-circuiting: every field has to be tokenized. Q1b, a grouping on the first variable-length field, allows partial short-circuiting. Q1c, a sum on a fixed-length field, allows full short-circuiting.

Q1a, Q1b, and Q1c are run on data compressed in each of the 6 field encodings and 4 tuple encodings. Figure 2.3 plots these numbers for the Sun. For comparison, Table 2.4 lists the numbers on Sun and Power-5 for Q1a. Note that tuple encodings that embed lengths for the fixed-length (0V or zero-b/B) field codings are not considered. For the purpose of this experiment, the parallelism level is set to a natural one for the architecture: 32 threads and 1 interleave for the 1-issue, 8-core-32-thread Sun, 2 threads and 2 interleaves for the 5-issue, 2-core Power.

In each plot, the y axis represents the scan speed in ns/tuple (calculated as (elapsed time)/(number of tuples)). The x axis enumerates the different field encodings, and the four curves represent the different tuple encodings. There are a few major observations to draw from this plot and table:
Figure 2.3 Scan Speed (ns/tuple) vs Compression Format as a function of the number of variable-length fields for queries that allow varying levels of short-circuiting (Q1a, Q1b, Q1c). Zero-B is for byte-aligned tuples, and zero-b is for bit-aligned tuples. The results are for a Sun-T1 with 32 threads and one interleave.
• Almost all scans are performed in under 10ns/tuple. On the Power 5 CPU, these same scans ran in 10-20 ns/tuple.

• Delta coding costs approximately an additional 2-4 ns/tuple during scan.

• As expected, variable-length fields reduce scan bandwidth. The slope of the curves indicates the cost for each variable-length field. It is about 1 to 2 ns per variable-length field for query Q1a, which cannot use short circuiting because of the select on the last attribute. But queries Q1b and Q1c, which can have short-circuiting, have much smaller slopes for the length-embedded formats append+len and delta+len (in Q1c, the time taken is almost independent of the number of variable length fields).

• The savings from byte alignment are negligible and not always present.

### Table 2.4 Scan Speed (ns/tuple) for Q1a on Power-5 and Sun-T1

<table>
<thead>
<tr>
<th></th>
<th>Power-5 (2 threads, 2 interleave)</th>
<th></th>
<th>Sun-T1 (32 threads, 1 interleave)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0V (byte)</td>
<td>0V (bit)</td>
<td>1V</td>
</tr>
<tr>
<td>A</td>
<td>4.00</td>
<td>4.00</td>
<td>5.36</td>
</tr>
<tr>
<td>AL</td>
<td></td>
<td></td>
<td>7.33</td>
</tr>
</tbody>
</table>

### 2.5.3 Predicate Evaluation Performance

Queries Q2 and Q3 study the cost of predicate evaluation as part of the scan. Q2 is a query with two equality predicates: one on a fixed-length field and one on a variable-length field. As before, the scan speed for each format is plotted, using 32 threads and 1 interleave (Figure 2.4). This plot is similar to the previous plots, except that the length embedded formats take the same
time regardless of how many variable-length fields are used. This is due to short circuiting, but the short circuiting is of a different nature than it was for Q1c in Figure 2.3. There, Q1c short circuited because it did not use the variable-length fields. Here, Q2 uses the last variable-length field, but it still short circuits because of the selective predicate.

Q3 has a range predicate that is a between predicate on a variable-length field. As Figure 2.5 shows, the performance is almost identical to that of the pure scan, except every number is 1-2ns/tuple higher. This represents the cost of evaluating a range predicate on a canonical Huffman coded field.

In our experiments, compilation (with the highest level of optimization) took on the order of a few seconds (1.5s-10s on the 1GHz Ultrasparc T1). This overhead is negligible for typical data warehouse queries that can run for hours. For shorter queries, we could compile multiple queries at the same time to amortize the cost of invoking the compiler, or directly output machine code. In the future we could consider generating the scan program in two compilation units, one with the inner loop (Fig 2) and one with the outer loop. We could then compile the compilation unit containing the inner loop with -O5 and use -O1 for the outer loop.

### 2.5.4 Value of Parallelism

Processors now provide multiple forms of parallelism, and the data is partitioned in order to try to take advantage of as much of it as possible. The Sun-T1 has 8 cores, 32 threads, and is single-issue; thus, interleaved parallelism is not applicable, but thread parallelism is. Figure 2.6 plots the scan speeds in tuples/ns as a function of the number of threads used, for some of the compression formats. The main observation is that every format, including the ones that do difference coding, scales up very nicely. The speedup is almost linear until 8 threads. This shows that splitting tuples into compression blocks works; even though processing of difference coded tuples within a block is serialized, there is good parallelism across blocks.
Figure 2.4 Scan Speed (ns/tuple) for Q2: one equality predicate on a fixed-length field and one on a variable-length field on the Sun-T1 with 32 threads and one interleave.

Figure 2.5 Scan Speed (ns/tuple) for Q3: 2 range predicates on the same variable length field on the Sun-T1 with 32 threads and one interleave.
Figure 2.6 Scan Bandwidth on the Sun-T1 for a scan (Q1b:top) and a predicate on a variable-length field (Q3:bottom)
2.5.5 Discussion: Which Format is Better?

Having seen the performance of a variety of compression formats, we are tempted to make judgments about which format is better. Our results have shown a clear trade off: the Huffman coded and difference coded formats that compress better generally scan more slowly.

But by keeping all our data sets in memory, we have expressly chosen to measure the CPU speed of the scan. For example, from Figure 2.3, the scan on the zero-B.Append (byte aligned, non difference coded) format runs at 4ns/tuple. But to actually realize this speed, the I/O and memory subsystem must flow 250M tuples/second into the processor.

We now plot the combined CPU and memory bandwidth requirements of these formats. Figure 2.7 plots the data from Figures 3, 4 and 5, in a different representation. The plot places six tuple formats on the x-axis (marked by vertical lines), scaled to their compressed sizes. For example, there’s a triangle for query Q1c at (19.48, 5.1): a scan speed of 5.1 cycles/tuple with a size of 19.48 bits/tuple (the 3V.DL format). The presented compression formats are either bit-aligned or have 1-3 variable-length fields. They are all delta coded, and some also have their length embedded. None of the appended tuple formats appear on the graph, since their compressed sizes are larger than 40 bits/tuple and thus require too much memory bandwidth to be interesting for this discussion.

Which of these formats to choose depends on how balanced the system is, in terms of bandwidth into the processor and the speed of the processor. On the same plot, we have also drawn dashed lines of the instructions per tuple available under different system configurations. For example, a main-memory DBMS on a 4-core server might run at 4x3 GHz, but be able to access its memory at 6GB/s. For the main-memory DBMS, any format to the left of its dashed line is a viable configuration – formats to the right will not be able to get tuples into the processor fast enough. A system designer has to choose the right compression format by plotting the dashed line of instructions/bit available on her system, and then picking a format close to this line.
Figure 2.7  CPU cost vs Memory/ I/O Bandwidth cost for chosen formats on the Sun-T1 with 32 threads and one interleave. Formats larger than 40 bits/tuple or that are dominated in both cycles/tuple and bits/tuple are omitted.
2.6 Conclusion and Future Work

This work begins by reviewing myriad compression techniques, and discusses the granularities at which the compression can be performed: the column level, the tuple level, and the block level. The techniques range from the simple, such as domain encoding, which always results in fixed-length codes, to Huffman coding, which results in variable-length codes, to delta coding, which works on sequences of tuples. Each of the different techniques has a different cost and effect on scan time. We also introduced fixed-length quantized codes and a length lookup table to decrease processing time.

Currently, the level of compression in a system is fixed, and cannot be easily modified to reflect the system’s abilities. In order to study the effects of compression at the CPU level across different systems, we created a scan generator, which takes compression, tuple representation, query and system parameters and outputs a custom scan C file. This custom scan file provides code for a fast scan, and may or may not include short-circuiting, interleaving, threading, and fast access to Huffman-coded fields. We believe that eventually database administrators will be able to use a physical format optimizer to tailor data sets to their computational environments and workloads.

All the results have shown that the Huffman coded and delta coded formats compress better but normally take more CPU time. However, these results do not include I/O time. When I/O and memory subsystem times are also included in the decision, the optimum format to choose becomes less clear. If a physical format optimizer or system administrator had this information and a fast scan generator, they could make a more informed choice as to the best way to store the data.
Chapter 3

Read-Optimized Databases, In Depth

In the previous chapter, we presented an extensive study of database scans over tables stored in a compressed row store format. While a number of recent papers have shown the benefits of column stores over row stores, the research comparing the two in an “apples-to-apples” way has left a number of unresolved questions. In this chapter, we first discuss the factors that can affect the relative performance of each storage paradigm. Then, we choose points within each of the factors to study further. Our study examines five tables with various characteristics and different query workloads in order to obtain a greater understanding and quantification of the relative performance of column stores and row stores. We then add materialized views to the analysis and see how much they can help the performance of row stores. Finally, we examine the performance of hash join operations in column stores and row stores.

3.1 Introduction

Recently there has been renewed interest in storing relational tables “vertically” on disk, in columns, instead of “horizontally”, in rows. This interest stems from the changing hardware landscape, where processors have increased in speed at a much faster rate than disks, making disk bandwidth a more valuable resource.

The main advantage to vertical storage of tables is the decreased I/O demand, since I/O is an increasingly scarce resource and most queries do not reference all the columns of a table. Vertical data storage has other advantages, such as better cache behavior [15, 58] and reduced storage overhead. Some argue that column stores also compress better than row stores, enabling the columns
to be stored in multiple sort orders and projections on disk for the same amount of space as a row store, which can further improve performance [58].

A vertical storage scheme for relational tables does come with some disadvantages, the main one being that the cost of stitching columns back together can offset the I/O benefits, potentially causing a longer response time than the same query on a row store. Inserting new rows or deleting rows when a table is stored vertically can also take longer. First, all the column files must be opened. Second, unless consecutive rows are deleted, each delete will incur a disk seek. Updates have similar problems: each attribute value that is modified will require a seek.

Traditional row stores store the tuples on slotted pages [51], where each page has a slot array that specifies the offset of the tuple on the page. The advantages to this paradigm are that updates are easy, and queries that use most or all of the columns are fast. The disadvantage is that, since most queries do not use all columns, there can be a substantial amount of “wasted” I/O bandwidth. Slotted pages also result in poor cache behavior [11] and are less compressible, since each tuple is stored individually and has its own header.

However, several recent studies [52, 29] (and the previous chapter) have demonstrated that row stores can be very tightly compressed if the tuples are stored dense-packed on the page without using slot arrays. While this format causes row stores to lose their advantage of easy updatability, the change can save substantial amounts of I/O bandwidth, which is where they often lose compared to column stores. [52, 29] also examine skewed data sets in row stores and use advanced compression techniques, such as canonical Huffman encoding [30], to achieve a degree of row store compression very close to that of the entropy for the table.

The confluence of these two ideas, column stores and advanced compression in row stores, brings us to the central question of this chapter: How do the scan times for row and column stores compare when both tables are as tightly compressed as possible? In other words, can row stores compete with column stores for decision-support workloads that are dominated by read-only queries?

“Performance Tradeoffs in Read-Optimized Databases,” was a first step toward answering this question [28]. This chapter provides an initial performance comparison of row stores and column
stores, using an optimized, low-overhead shared code base. Additionally, the tuples for both storage paradigms are stored dense-packed on disk, a necessity for truly obtaining an apples-to-apples performance comparison.

[28] studies both wide and narrow tables using the uniformly distributed TPC-H [22] Lineitem and Orders tables. The two formats are compared without compression for Lineitems, and both compressed and uncompressed for Orders. The Orders table is compressed using bit-packing, dictionary encoding and delta coding, where appropriate. All queries are of the form: “Select column_1, column_2, column_3, ... from TABLE where predicate(column_1)” with a selectivity of 0.1% or 10%.

The results from [28] that are most relevant to this chapter are that:

- At 10% selectivity, when half of the columns are returned, a row store will be faster than a column store only if the tuples are less than 20 bytes wide and the system is CPU-constrained.

- The CPU time required for a column store to execute a scan query is very sensitive to the number of columns returned and the selectivity factor of the query.

- The time required for a row store to process a scan query is relatively insensitive to the number of columns being returned or the selectivity of the predicate.

- Column stores can be sensitive to the amount of processing needed to decompress a column.

However, these results left a number of questions unresolved. First, the attribute values in each column of the TPC-H tables are uniformly distributed. The results presented in [52, 29] demonstrate that, if the distribution of values in a column is skewed, even higher degrees of compression can be obtained. Thus, in this chapter we explore non-uniform value distributions and the impact of advanced compression techniques.

Additionally, [28] considered only a very limited set of queries. First, no queries with selectivity factors greater than 10% were studied. Also, all of the queries considered in [28] consisted of a single predicate that was always applied to the first column of the table and only the left-most
columns of a table were returned. We believe that the relative performance of column and row stores is affected by (a) how many predicates the query has, (b) what columns the predicates are applied to, and (c) which columns are returned. A key focus of this chapter is to explore each of these issues.

Finally, we noticed that the row store performed very poorly compared to the column store for the Lineitem table, which has a long string attribute. While the string does not impair the performance of a column store if it is not used in the query, it disproportionately hurts the row store since that one attribute is the same size as the sum of the others. We hypothesized that if the string had been stored in a separate table, the performance of the two paradigms would have been more comparable. Hence, our study will also include materialized views of the row store.

While [28] was a first step at an “apples-to-apples” comparison of row stores and column stores in a common software framework, we want to achieve a greater understanding of the issues raised above. The aim of this work is to systematically study scans for compressed row and column stores. Specifically, we explore a large space of possible tables, compression methods, and queries, in which the results of [28] would fall as points in this space. This chapter’s contributions are to:

- Provide a more in-depth discussion of the factors that affect the scan times for read-optimized row stores and column stores, and to choose points within those factors to study further.

- Quantify the effect of (a) more qualifying tuples, (b) additional predicates, and (c) tables that do not have uniform data distributions.

- Compare the performance of materialized views with the performance of column stores and row stores in this framework.

- Examine the effect a hybrid hash join has on column stores and row stores, when using early materialization of tuples in the join.

Section 3.2 discusses other related work. Sections 3.4 and 3.5 discuss the overall search space and how points within it were selected for evaluation. The remaining sections provide implementation details and results. We conclude in Section 3.7 with a short discussion of our work and how the results might be applied.
3.2 Related Work

3.2.1 Database Compression Techniques

Most research on compression in databases has assumed slotted-page row stores. Two papers that have not made this assumption, [29, 52], were mentioned above. Both papers also used more processing-intensive techniques to achieve compression factors of up to 8-12 on row stores. “Superscalar RAM-CPU Cache Compression” [66] and “Integrating Compression and Execution in Column-Oriented Database Systems” [8] examine compression in column stores. Zukowski finds that compression can benefit both row stores and column stores in I/O-restricted systems and presents algorithms to optimize the effective use of modern processors. [8] considers various forms of compression and the effects of run lengths on the degree of compression achieved. The main conclusion related to this study is that RLE and dictionary encoding tend to be best for column stores.

3.2.2 Vertical Storage

Very few papers directly compare row stores to column stores in an apples-to-apples way. [28] is one, and another is “Comparison of Row Stores and Column Stores in a Common Framework” [26]. Two contributions of this paper are the idea of a super-tuple and column abstraction. The authors note that one reason row stores do not traditionally compress as well as column stores is the use of the slotted-page format. The paper introduces the idea of super-tuples, which store all rows on a page with just one header, instead of one header per tuple and no slot-array. Column-abstraction avoids storing repeated attributes multiple times by adding information to the header. The paper compares results for 4-, 8-, 16-, and 32-column tables, however, it focuses on uniformly distributed data and examines trends within column stores, row stores and super-tuples when returning all columns and all rows. While those results are interesting, we are more interested in looking at the general case where a query can return any number of columns.

One of the first vertical storage models was the decomposition storage model (DSM) [19], which stored each column of a table as pairs of (tuple id, attribute values). Newer vertical storage
systems include the MonetDB/X100 [15] and C-Store [58] systems. MonetDB operates on the columns as vectors in memory. C-Store differs from DSM in that it does not explicitly store the tuple id with the column. Another novel storage paradigm is PAX [12], which stores tuples column-wise on each disk page. This results in better L2 data cache behavior, but it does not reduce the I/O bandwidth required to process a query. Data Morphing further improves on PAX to give even better cache performance by dynamically adapting attribute groupings on the page [27].

### 3.3 Factors Affecting Performance

In this chapter, we consider four factors that can have a significant effect on the relative performance of row stores and columns stores:

1. The width of a table (i.e. number of columns), the cardinality of each column (i.e. the number of unique attribute values in the column), and the distribution of these values (uniform or skewed).

2. The compression techniques employed.

3. The query being executed, including the number of predicates and which columns the query returns.

4. The storage format (i.e. slotted pages or super-tuples).

These four factors combine to produce a very large search space. In the following sections, we describe each factor in additional detail. In Section 3.5, we explain how the search space was pruned in order to make our evaluation feasible while still obtaining a representative set of results.

### 3.3.1 Data Characteristics

Each relational table can have a range of characteristics. First, and foremost, the number of columns and rows in a table directly affects the time required to scan a table. Second, the type of each attribute column (i.e. VAR CHAR or INT) determines how the data would be stored when uncompressed, and the distribution of values within the column (such as uniform or skewed) affects
how well the column’s values can be compressed. The number of unique values stored within the column, or column cardinality, also affects the column’s compressibility.

To illustrate, imagine a table with a single column of type INT which can only assume one of the four values: 1, 2, 3, or 4. The column cardinality for this column is 4. If the values are distributed equally within the column (i.e., each value has the same probability of occurring), then the column’s distribution is termed uniform, otherwise, the distribution is termed skewed. For a row-store with a slotted page format, this column would most likely be stored as a 32-bit integer. However, only two bits are actually needed to represent the four possible values. The distribution of the values and whether or not the table is sorted on this column will determine if the column can be stored in, on average, less than two bits per row by using compression.

3.3.2 Compression

Many different types of compression exist in the literature, from light-weight dictionary coding, to heavy-weight Huffman coding. Since a full evaluation of compression techniques is outside the scope of this chapter, based on the results in [29], we focus on those techniques that seem to generally be most cost-effective. These techniques are bit-packing, dictionary coding, delta coding, and run-length encoding (RLE).

As demonstrated in the example above, bit packing uses the minimum number of bits to store all the values in a column; for example, two bits per attribute value are used instead of the 32 normally needed for an INT. This space savings means that bit-packed attributes (whether in a column or row) will not generally be aligned on a word boundary. With the current generation of CPUs, that cost has been found to be negligible [8, 29].

Dictionary coding is another intuitive compression technique in which each attribute value is replaced with an index into a separate dictionary. For example, the months of the year (“January” to “December”) could be replaced with the values 1 to 12, respectively, along with a 12-entry dictionary to translate the month number back to a name. Although dictionaries are often useful, they can sometimes hurt performance, for instance, if they do not fit into the processor’s L2 data cache. Dictionaries should also not be used if the index into the dictionary would be bigger than
the value it is replacing, or if the size of the un-encoded column is smaller than the size of the encoded column plus the size of the dictionary.

Delta coding stores the difference between the current value and the previous value. To delta encode a sequence, first the sequence is sorted and the lowest value is stored. Then, each difference should be stored in two parts: the difference itself starting at the first “1” in the bit representation of the difference, and the number of leading zeroes. For instance, consider the sequence 24, 25, 27, 32. The bit representation for each is 011000, 011001, 011011, 100000, respectively. The differences between subsequent values in the sequence are 000001, 000010, and 000101. Thus, the sequence to encode would be 24, (5, “1”), (4,“10”), (3,“101”), which would be 011000 101 1 100 10 011 101. The number of leading zeroes should be encoded as a fixed-width field, but the difference will generally be variable length. To simplify decoding, the number of leading zeroes should be stored before the difference. Delta coding can be performed both at a column level, and at a tuple level, provided that each column of the tuple has been bit packed. Delta coding should not be used on unsorted sequences or when the encoding for the number of leading zeroes plus the average size of the differences in bits is bigger than the non-delta-coded value.

Run-length encoding (RLE) transforms a sequence into a vector of <value, number of consecutive occurrences (runs)> pairs. For instance, the sequence 1, 1, 1, 1, 2, 2, 4, 4, 4 would become <1,4>, <2,2>, <4,3>. RLE compresses best if the sequence has first been sorted, and if there are many long runs.

Bit packing, dictionary coding and run-length encoding are all light-weight compression schemes. They can lead to substantial space and I/O savings while incurring very little extra processing. Delta coding can provide extremely good compression for the right type of data, but requires much more processing to decode [29].

Generally, the row store and column store versions of the same table should be compressed using different techniques. For instance, RLE is often a good scheme for column stores but not for row stores, since it is rare to have multiple consecutive rows that are identical.
3.3.3 Query Characteristics

The third factor we considered was the characteristics of the queries to be used for evaluating the relative performance of the column and row store configurations, including the number of predicates and the columns to which the predicates are applied, the number of output columns and which ones, and the selectivity of the selection predicate(s). To simplify the search space somewhat, we primarily considered scan queries. Each query materializes its output as un-encoded row-store tuples, but these tuples are not written back to disk. The number of predicates and the number of output columns affect the number of CPU cycles needed for the query. In this chapter, a predicate is a Boolean expression of the form “Col_i ≥ Value.” For queries with more than one predicate, predicates are “anded” together.

The type of each column can also have a significant effect on scan time, since different types can consume varying amounts of CPU and I/O time. A column of CHARs will take four times less I/O than a column of INTs, and a well-compressed RLE column of CHARs will take even less, while scanning a delta-coded column may consume a substantial amount of CPU time.

The output selectivity (i.e. the number of tuples returned), also impacts the scan time since materializing result tuples requires, at the very least a copy.

3.3.4 Storage

The two main paradigms used by the current generation of commercial database system products for table storage are row stores with slotted pages and dense-packed column stores. However, there are other possibilities, such as PAX [12], DMG [27], and column abstraction [26]. Column abstraction tries to avoid storing duplicate leading attribute values multiple times. For instance, if four tuples in a row have “1” in their first attribute, that knowledge is encoded in the first header, and the subsequent tuples do not store the “1.” We will refer to dense-packed row stores as “super-tuples” [26].
3.4 Narrowing the Search Space and Generating the Tables

Once the overall parameters were identified, they had to be narrowed down to a representative and insightful set. The parameter space can be viewed as two sets: the set of tables (affected by the data characteristics, storage format, and compression) and the set of queries.

3.4.1 Table Parameters

We elected not to use TPC-H tables since we wanted more control over all aspects of the tables. Instead, we devised five tables:

1. Narrow-E: This table has ten integer columns and 60 million rows. The values in column i are drawn uniformly and at random from the range of values \([1 \text{ to } 2(2.7 * i)]\): column 1 from the values \([1 \text{ to } 6]\); column two from \([1 \text{ to } 42]\); column three from \([1 \text{ to } 274]\); column four from \([1 \text{ to } 1,782]\); column five from \([1 \text{ to } 11,585]\); column six from \([1 \text{ to } 75,281]\); column seven from \([1 \text{ to } 489,178]\); column eight from \([1 \text{ to } 3,178,688]\); column nine from \([1 \text{ to } 20,655,176]\); and column ten from \([1 \text{ to } 134,217,728]\).

2. Narrow-S: This table is similar to the table in Narrow-E (ten integer columns and 60 million rows) but the values in each column are drawn from a Zipfian distribution with an exponent of 2 instead of uniformly at random from the given ranges.

3. Wide-E: This table is similar to Narrow-E, except it has 50 columns instead of 10 and 12 million rows instead of 60 million. Thus, the uncompressed sizes of the two tables are the same. The values for column i are drawn uniformly and at random from the range \([1 \text{ to } 24+(23/50*i)\]).

4. Narrow-U: This table has ten integer columns and 60 million rows. The values in each column are drawn uniformly and at random from the range \([1-100 \text{ million}]\).

5. Strings: This table has ten 20-byte string columns and 12 million rows. No compression is used on this table.
We initially studied a wider range of tables, but we found that these tables provide the most interesting and, in our opinion, representative results.

We limited our study to predominantly integer columns, since they are common and generalize nicely. However, we did decide to study one table comprised solely of strings. Strings tend not to compress as well as integers and at the same time are wider than integers, so they are an important point to study.

The tables are stored on disk in either row store or column store format, with both using super-tuples and no slot array. This storage method results in read-optimized tuples.

Since there are so many different compression techniques, it would be impossible to implement and test all of them in a timely manner. Thus, this study compresses each table once for each storage format (row and column) using the compression technique that optimizes the I/O and processor trade-offs. While Huffman encoding could provide better compression for the Narrow-S table, it often results in worse overall performance due to the extra CPU cycles it consumes decoding attribute values. How the tables are generated and compressed is presented in the next section.

### 3.4.2 Table Generation and Compression

First, an uncompressed row-store version of each of the five tables was generated. Then, each table was sorted on all columns starting with the left-most column in order to maximize the run lengths of each column. From this table, projections of each column were taken to form the column-store version of the table. The value of each attribute of each column is a function of a randomly generated value, the column cardinality, and the column’s distribution. For the table Narrow-S, a traditional bucketed implementation of Zipfian took prohibitively long with column cardinalities greater than 1000 so a faster algorithm that produces an approximation of Zipfian results was used [1].

The compression technique(s) used for each configuration of each table is shown in Table 3.1 below. Dictionaries are per page. That is, the dictionary for the attribute values (whether organized as rows or columns) on a page is stored on the same page.
Table 3.1 Type of compression used for each table.

<table>
<thead>
<tr>
<th>Table</th>
<th>Row Store Compression</th>
<th>Column Store Compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Narrow-E</td>
<td>All columns bit-packed</td>
<td>Columns 1-3 RLE,</td>
</tr>
<tr>
<td></td>
<td>Columns 1-5 delta encoded</td>
<td>Columns 4-10 bit-packed</td>
</tr>
<tr>
<td>Narrow-S</td>
<td>Columns 1-6 bit packed</td>
<td>Columns 1-6 RLE,</td>
</tr>
<tr>
<td></td>
<td>Columns 7-10 dictionary encoded</td>
<td>Columns 7-10 dictionary encoded</td>
</tr>
<tr>
<td>Wide-E</td>
<td>All columns bit-packed</td>
<td>Columns 1-4 RLE,</td>
</tr>
<tr>
<td></td>
<td>Columns 1-8 delta encoded</td>
<td>Columns 5-50 bit-packed</td>
</tr>
<tr>
<td>Narrow-U</td>
<td>All columns bit-packed</td>
<td>All columns bit-packed</td>
</tr>
</tbody>
</table>

Traditionally, RLE uses two integers: one for the value, and one for the run length. However, this can easily result in sub-optimal compression, particularly when the value range for the column does not need all 32 bits to represent it. As a further optimization, we store each RLE pair as a single 32-bit integer, with n bits used for the value and 32-n bits for the run length. For example, consider the last column of Narrow-E, whose values range from 1 to 134 million. In this case, 27 bits are needed for the value, leaving 5 bits to encode the run length. If the length of a run is longer than 31, the entire run cannot be encoded into one entry (32 bits), so multiple entries are used. This optimization allows us to store the RLE columns in half the space.

For the table Narrow-S, columns 7-10 are dictionary encoded for the row store. Each column is individually dictionary-encoded, but, in total, there can be no more than 2,048 (2^11) dictionary entries shared amongst the four columns. This requirement means that each of the attribute values is encoded to 11 bits, but allows the number of entries for each individual column’s dictionary to vary from page to page.

To find a reasonable dictionary size, we estimated how many compressed tuples, t, would fit on a page, then we scanned the table to calculate the dictionary size needed to fit t tuples per page. The reasoning behind this approach is that the page size must be greater than or equal to the size of the dictionary and all of the tuples on that page. A 2,048 entry dictionary is 8KB, so for a 32KB page, there is room for 24KB of data. So, if the data are expected to compress to 8 bytes per tuple (assuming each dictionary encoded column is 11 bits), 3K tuples will fit on the page. If, after scanning the data, it is found that a 2,048 entry dictionary would be too small, the dictionary size
should be doubled and the analysis redone. For the best dictionary compression possible, analysis would be performed for every page, but we were satisfied with the compression we obtained using this simpler method.

For the column store version of Narrow-S, columns 7-10 are also dictionary encoded. Since each column is stored in its own file, the dictionary does not need to be shared. However, a larger dictionary is needed; in this case, we allow a maximum of 4,096 dictionary entries per page, which encodes to 12 bits per value.

### 3.4.3 Query Parameters

In order to understand the base effects of the different tables and queries, most of the queries we tested have one predicate. However, a few have three predicates in order to study the effect of queries with multiple predicates on response time. Each query is evaluated as a function of the number of columns included in the output table: 1, 25% of the columns, 50% of the columns, and all columns. The selectivity factor is varied from 0.1% to 50%.

### 3.4.4 Scan Generation

The scan generator takes four inputs: the schema definition for the input table (Narrow-E, Narrow-S, Narrow-U, Wide-E, Strings), the desired selectivity factor, the total number of columns referenced, c, and the number of predicates, p. Since each column has different processing and I/O requirements, which columns are used, and the order in which they are processed, affects the execution time of logically equivalent queries. Thus, for each configuration of input parameters, the scan generator randomly selects the columns used to produce a set of equivalent queries. This differs from [28], where every query returned the columns in the same order, and the single predicate was always applied to the first column.

The scan generator first randomly picks c columns to include in the output table. The first p of those columns have predicates generated for them. The column store’s query is generated first. There is a different column scanner for each kind of compression used: an RLE scanner, a dictionary scanner, and a bit-packed scanner. The generator knows which scanner to choose
based on the table’s schema and column number (see Table 3.1). The inputs to the scanner are then picked, including any predicate values. The primary scanner inputs are the data file, that column’s offsets within the output tuple, and the number of bits needed to represent the encoded input column value. After all the columns have been processed, the generator outputs C++ code for the column store variant of the query.

Then, using the same set of columns, the code for the row store’s query is generated. Each of the five table types (see Section 3.5.1) must be scanned differently because of compression, and the generator outputs the minimum code necessary to decode the columns used in the query in order to save processing cycles.

Some desired output selectivities are difficult to obtain when queries are randomly generated. For example, assume that the input table is Narrow-E and that the query has a single predicate on the first column. In this case it is only possible to obtain selectivity factors that are multiples of 0.1666 since the first column only has six values that are uniformly distributed. Since the selectivity factor affects performance, we needed a way to obtain finer control over the selectivity factor of the query’s predicates. To do this, we modified how predicates are evaluated by also incorporating how many times the function itself has been called. For instance, if the desired selectivity is 10%, every tenth call to the predicate function will pass, regardless of whether the attribute value satisfies the predicate. Since selectivities are multiplicative, when there are multiple predicates, each predicate has a selectivity of desired selectivity(1/p), where p is the number of predicates.

3.5 Implementation Details

The results presented in this chapter are based on the same code-base as [28], which is provided online [2]. The scanners and page creation procedures were modified to allow for the different forms of compression. We first discuss why we chose this code-base, then the next two subsections discuss and summarize the salient features of the software. The last subsection gives the experimental methods.
3.5.1 Choice of Code-Base

We chose this code-base for a variety of reasons. First, our work follows on to that in [28]; thus, we use the same code so that a direct comparison can be made between our results and theirs. The code is also easy to understand and modify, and, most importantly, is minimal, so we can have more confidence that performance differences between the row store and the column store are fundamental, and not related to code overheads or quirks in the file system. Additionally, this experimental setup has passed the vetting process of reviewers.

The two main research column-store databases are C-Store and MonetDB [58, 15]. Both systems are available online, but they are heavier-weight, and we are trying to understand performance fundamentals. MonetDB uses a different processing model, where columns (or parts of columns) are stored in vectors in memory, whereas we assume the columns are disk resident. MonetDB proponents argue this main memory-based kernel can provide better performance than C-Store, but, to our knowledge, no direct comparison of the systems has been published.

3.5.2 Read-Optimized Pages

Both the row store and column store dense pack the table on the pages. The row store keeps tuples together, placed one after another, while the column store stores each column in a different file. The page size is 32 KB. In general, the different entries on the page are not aligned to byte or word boundaries in order to achieve better compression.

Each page begins with the number of entries on the page. The row or column entries themselves come next, followed by the compression dictionary (if one is required). The size of the compression dictionary is stored at the very end of the page, with the dictionary growing backwards from the end of the page towards the front. For the row store, the dictionaries for the dictionary-compressed columns are stored sequentially at the end of the page.

3.5.3 Query Engine, Scanners and I/O

The query engine and table scanners provide only a minimal set of functions. All queries are precompiled, and the query executor operates on 100 rows of the table at a time. The scanners
decode values, apply predicates and either project or combine the columns into an output tuple buffer.

The query scanner and I/O architecture are depicted in Figure 3.1. Since the row scanner is simpler to understand, it is explained first. The relational operator calls “next” on the row scanner to receive a block of tuples. The row scanner first reads data pages from disk into an I/O buffer, then iterates through each page in the buffer. The scanner always decodes the columns of the tuple that might be used, then applies the predicate(s). If the tuple passes the predicate(s), the uncompressed projection is written to the output tuple buffer. When the buffer is full, it is returned to the relational operator parent, which can print the tuples, write them to disk, or do nothing (for our experiments the parent operator simply tosses the output tuples). The relational operator then empties the buffer and returns it to the scanner.

The column scanner is similar to the row scanner, but must read multiple files: one for each column referenced by the query. Each column is read until the output tuple buffer is full (this buffer is discussed shortly); at that point, the read requests for the next column are submitted. Since predicates are applied on a per-column basis, columns are processed in order of their selectivity, most selective (with the fewest qualifying tuples) to least selective (the most qualifying tuples). Placing the most selective predicate first allows the scanner to read more of the current file before having to switch to another file, since the output buffer fills up more slowly.

For each attribute value that satisfies its predicate, the value and its position in the input file are written into the output buffer. The passing positions (pos list in Figure 3.1) are then input into the next column’s scanner, and that column is only examined at those positions.

The output tuple buffer holds 100 tuples. At this size, it can fit in the 32KB L1 data cache, even if there are dictionaries, for each of the five different tables. This buffer is used to reduce overhead. Instead of performing aggregate computation, outputting qualifying tuples, or switching from column to column for every individual tuple, these operations are batched. For our experiments, the qualifying tuples are simply materialized: they are not output to the screen or to disk. An output buffer size that is too small can lead to unnecessary overhead and poor performance by, for instance, writing to disk as soon as one tuple has been found to qualify. On the other hand, if
Figure 3.1 Query Engine Architecture [28].
the buffer is too big, it will fall out of the data cache and increase processing time. We ran scans with multiple buffer sizes and found 100 gave the best performance.

All I/O is performed through Linux’s Asynchronous I/O (AIO) library. The code’s AIO interface reads I/O units of 128KB (a user-set parameter), and it allows non-blocking disk prefetching of up to depth units. Data is transferred from the disk to memory using DMA and does not use Linux’s file cache. The code does not use a buffer pool, instead it writes the transferred data to a buffer pointed to by a program variable.

3.5.4 System and Experimental Setup

All results were run on a machine running CentOS v4.0 on a 2.4 GHz Intel Core 2 Duo processor and 2GB of RAM. The disk is a 320 GB 7200 RPM SATA Western Digital WD3200AAKS hard disk. We measured its bandwidth to be 70 MB/s. Runs are timed using the built-in Linux “time” function. For each combination of output selectivity and number of columns accessed, 50 equivalent queries were generated. The run times presented in Section 3.6 are the averages of those queries.

Sometimes we observed transient AIO errors, which caused a query to abort. When this happened, the run was terminated and restarted. We verified that runs that completed without errors returned the correct results. We also verified that running the same query multiple times in a row had negligible timing differences between runs.

The program has four user-defined variables: I/O depth, page size, I/O unit (scan buffer) size, and block (materialized tuple buffer) size. We use an I/O depth of 48; thus, each scan buffer can be prefetched up to 47 I/O units in advance. We use a larger page size than [28], but otherwise the program parameters are the same. We decided to use 32KB pages instead of 4KB pages since we think the larger size is more common in practice. However, [28] found that, as long as the page was not too small, the size did not significantly impact performance.
3.6 Results

The results of our experiments are presented in this section, beginning with the amount of compression we were able to obtain.

3.6.1 Amount of compression

The compression methods used to encode each table are listed in Table 3.1. Table 3.2 presents their compressed sizes. The compression factors achieved range from 1 to 3 if just variable length, bit aligned attributes are used (column 3) and from 2 to almost 6 when compressed (column 4). While techniques that compress the tables even more could have been used (e.g. Huffman encoding), we think the techniques presented are a good trade off.

This final compressed size is almost exactly the same for both the row store and the column store, i.e. the size of the compressed row store file is the same as the sum of the column file sizes for the column store. This result is very important since column store proponents argue column stores compress better than row stores, so the columns can be saved in multiple sort orders and projections for the same space as one row store [58]. Thus, for read-optimized row stores, this assertion is not true, even with aggressive column store compression.

Table 3.2 Total tuple sizes with and without compression.

<table>
<thead>
<tr>
<th>Table</th>
<th>Uncompressed</th>
<th>Bit-aligned</th>
<th>Compressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Narrow-E</td>
<td>40 Bytes</td>
<td>153 bits (20 Bytes)</td>
<td>17 Bytes</td>
</tr>
<tr>
<td>Narrow-S</td>
<td>40 Bytes</td>
<td>153 bits (20 Bytes)</td>
<td>7 Bytes</td>
</tr>
<tr>
<td>Wide-E</td>
<td>200 Bytes</td>
<td>811 bits (102 Bytes)</td>
<td>100 Bytes</td>
</tr>
<tr>
<td>Narrow-U</td>
<td>40 Bytes</td>
<td>270 bits (34 Bytes)</td>
<td>34 Bytes</td>
</tr>
</tbody>
</table>

3.6.2 Effect of selectivity

In Figure 3.2 we explore the impact of selectivity factor as a function of the number of columns returned for table Narrow-E. The x-axis is the number of columns returned, and the y-axis is the elapsed time. For both the column store (C-%) and row store (R-%) each line corresponds to a
different output selectivity, with one, three, five or ten columns returned, and one predicate. Each data point is the average of 50 randomly generated queries, as described in Sections 3.4.4 and 3.5.4. The time to scan the uncompressed row store is also included (R-Uncomp) to illustrate how much time compression can save. Additionally, this graph presents error bars of plus and minus a standard deviation; however, the standard deviations are often quite small, so the error bars are difficult to see. For the row stores, all but the selectivity of 50% line have been omitted, since the response time for the compressed row store was not significantly affected by the selectivity factor of the query. The C-0.1% and C-1% response times are also almost exactly the same.

The column store is faster than the row store when eight of the columns are returned with selectivity factors of 0.1% and 1%; when five of the columns are returned with a selectivity factor of 10%; when two of the columns are returned with a selectivity of 25%; and basically never at a selectivity of 50%. Further, for this table configuration, the best speedup of the column store over the compressed row store is about 2, while the best speedup for the compressed row store over the column store is about 5.

Next, we turn to investigating the factors underlying the performance of the two systems at a selectivity factor of 10%. Figure 3.3 presents the total CPU and elapsed times for both the row and column store. Both the CPU and elapsed times are constant for the row store as the number of columns returned is increased, since the run is primarily disk-bound. The results for the column store are substantially different, as the CPU time consumed increases as more columns are returned. This increase comes from two main sources: more values must be decoded, and more tuples must be stitched back together (materialized) using some form of a memory copy instruction. At a selectivity factor of 0.1%, the CPU cost of the column store is constant, since very few tuples must be materialized, and, at 50% selectivity, both the column store and the row store become CPU-bound. While the number of columns returned definitely has a large effect on column store scan times, the selectivity of the predicate is also an extremely important factor.
Figure 3.2 Elapsed time for Narrow-E with various selectivity factors.

Figure 3.3 Elapsed time and CPU time for Narrow-E with 10% selectivity.
3.6.3 Effect of Skewed Data Distribution

To study the effect of a skewed data distribution, we repeated the experiments on table Narrow-S that we had performed on Narrow-E. Narrow-S allows more columns to be run-length (or delta) encoded, and made dictionary compression worthwhile for the columns with larger column cardinalities (number of different values in the column). Using these techniques, the row store went from averaging 17 bytes per tuple to 7 bytes per tuple (recall that the uncompressed tuple width is 40 bytes). The total size of the compressed column store tuple is the same as that for the row store tuple; each column compressed to less than two bytes per attribute, with the first attributes averaging just a few bits. The selectivity graph is presented in Figure 3.4. Error bars are also present on this graph, but the standard deviation is again small enough that they cannot be seen. Again each data point represents the average of 50 different randomly generated queries in which both the columns returned and the column to which the predicate is applied are selected at random. The difference between the elapsed times for the column store runs with 0.1% and 1% selectivity factors are negligible. At 0.1% and 1% selectivity, the column store (C-0.1%,1%) is faster than the row store (R-0.1%,1%) when seven columns are returned, and at 10% selectivity, the column store is faster than the row store when three columns are returned. At a selectivity factor of 50% the row store is always faster. For this table, the elapsed time for the row stores is affected by the selectivity factor since very little bandwidth is needed. Thus, the CPU time consumed dominates the overall execution time for the query for both column stores and row stores. Those queries with higher selectivity factors require more computation, so they have longer run times. For this table, the performance of the row store is very competitive with the column store largely due to the significant decrease in the amount of I/O it performs.

3.6.4 Wide Tables

Our next experiment was conducted using the table Wide-E. Wide-E has fifty integer (4 byte) columns, whose column cardinalities increase from left to right (see Table 1 for details). After bit compression, each tuple averages 100 bytes. To keep the total uncompressed table size the same as for Narrow-E, Wide-E has only 12 million rows (instead of 60 million). The time to scan the
uncompressed table is shown in the R-Uncomp line. Again, the elapsed time for the row store is dominated by its disk I/O component and the selectivity factor of the query, and the number of columns returned does not have an observable effect on the response time. For the column store, the response times with either a 0.1% or 1% selectivity factor are essentially the same. The row store is faster than the column store when 85% of the columns are returned with 0.1% or 1% selectivity factors; when returning 66% of the columns at 10% selectivity, and with 25% of the columns at 50% selectivity.

Overall, the graphs of Figure 3.5 and Figure 3.2 are very similar, however, the row store does not compress as well (and hence takes longer to scan), so there is a shift in the crossover points between the two storage paradigms. Because the slopes of the column store lines are not very steep, how well the row store compresses is a critical component in determining its performance relative to that of the column store. It should also be noted that many more columns are being returned than with the narrow table Narrow-E.
If fewer than twelve columns are needed, the row store is always slower than the column store for this configuration.

![Graph](image)

Figure 3.5 Elapsed time for Wide-E at various selectivity factors.

### 3.6.5 Effect of Column Cardinality

Figure 3.6 presents the elapsed times for the experiments with Narrow-U. Like Narrow-E, this table has ten columns, but the values in each column are drawn uniformly and at random from a range of 1 to 100,000,000. The average compressed size of each tuple is 34 bytes, double that of the tuples in Narrow-E. The larger table and column sizes result in a significant increase in the response times that we measured for the two systems due to the additional I/O operations performed. The crossover points of the two systems are very similar to those for the wider and shorter table discussed in the previous section (see Figure 3.5).
3.6.6 Additional Predicates

The previous results have shown that the selectivity of the predicate can have a substantial effect on the relative performance of the row store and column store. We next consider how adding additional predicates affects the relative performance of the two systems by considering queries with three predicates (instead of one) and with selectivity factors of 0.1% and 10%. Table Narrow-E was used for these tests and the results are presented in Figure 3.7.

For each selectivity, we ran two sets of experiments. In the first set of experiments, the first, second, and third columns (x% left 3 preds) are used as the predicate columns (Figure 3.7 (top)). In the second, the columns for the three predicates were selected at random (x% random 3 preds) (Figure 3.7 (bottom)). The “C-x% 1 pred” results are taken from the experiments presented in Figure 3.2. Since the 1 predicate point only uses one column, there is a predicate only on that column. The row results are only shown for the 10% selectivity factor.

We elected to draw predicates both from randomly selected columns and from the left three columns because we expected that the response time when the left hand columns were used would
be better than when randomly selected columns were used. Since the table is sorted on these columns, the runs in the left-most columns are longer. Hence, they compress better than the other columns. Our results verify that hypothesis. Our results also indicate that additional predicates can significantly affect the relative performance of the two systems. For instance, the 10% selectivity crossover is at five columns for the one-predicate case, but shifts to two columns when there are three predicates. The results are less stark for the 0.1% selectivity case since it requires so much less computation to begin with, but it still shifts the crossover from eight columns to seven.

These three-predicates results represent a worst-case scenario, as the selectivity is evenly divided between the columns. The results would be closer to the one predicate case if the first predicate had been highly selective. Thus, “real” workload results would probably fall somewhere in between the two points. However, the fact remains that increasing the number of predicates can have a significant impact on the performance of a column store.

### 3.6.7 Effect of “Wider” Attributes

We examine the impact of wider attributes by studying a table with ten 20-byte attributes. No compression was used for either the column store or the row store as we wanted to simulate the worst-case scenario for both systems. For this configuration, I/O is the dominant factor in the response times of both systems. Thus, the elapsed time of the column store is only slightly affected by the different selectivity factors, as can be seen in Figure 3.8.

To put this table’s size in perspective, each width of column of this table is about the same as the compressed tuples in the Narrow-E table, and is about three times the size of the tuples in the Narrow-S table. Each tuple is twice as big as the compressed tuples in the Wide-E table.

### 3.6.8 Materialized Views

There is, of course, an intermediate form between row stores and column stores: materialized views. We implemented two sets of materialized views for the Narrow-E table. One set groups the ten attributes into five pairs: columns 1&2, 3&4, 5&6, 7&8, 9&10; the other groups the attributes
Figure 3.7 Three predicates on left-most columns (top) or random columns (bottom).
Figure 3.8 Elapsed time for String with various selectivity factors.
into two groups of five: columns 1-5 and 6-10. We modified the column scanner to return an array of values instead of one value. Only columns used in the query are decoded.

The benefit of materialized views depends greatly on the amount of correlation between columns exhibited by the query workload. Commercial products often provide guidance in forming views for row stores, and for column stores, Vertica has a Database Designer to aid in selecting which projections and sort orders would provide the best overall performance for a query workload [61]. However, since we are using a random scan generator, we cannot rely on these automatic designers. Instead, we created the views, and then varied the correlation between the columns in randomly-generated workloads to find the benefit.

We looked at four different amounts of column correlation for each set of materialized views: 100%, 75%, 50% and none (independent). For the set of views where there are two views of five columns each, the scan generator was set up so that one column was picked at random. Then, when there are five or fewer columns used, there is a Correlation% chance that there is only one view needed. For the set of five views of two columns each, first one column is drawn at random. Then, there is a Correlation% chance that the other column from that view is used. If the correlation test fails, the second column from that view is not used for that query, unless all five of the views are used and more columns are needed, and another column is randomly drawn. All columns are drawn without replacement. In the “none” correlation case, all columns are drawn independently and at random, as in the earlier experiments.

Figure 3.9 provides the results for having two groups of five (top) or five groups of two (bottom) materialized views on the Narrow-E table. The queries have 1% selectivity, which is a selectivity where the column store’s performance dominates. The row and column store results are those from earlier sections. We gathered materialized view results for returning up to six, then ten columns. We took more data points since the results are not smooth lines: they have stair-step performance due to the correlations between the columns. When the correlation between columns is high, the performance is similar to that of joining multiple small row stores. When there is no correlation between columns, the materialized views do not provide a large benefit for row stores compared to column stores, since multiple views must be joined, and reconstructing those tuples can be costly.
In fact, the materialized views with five columns only outperform the row stores when three or fewer columns are returned, assuming no correlation. If more than five columns are used, the row store should be used due to the cost of stitching the two views together. The two column views outperform the row store when five or fewer columns are returned, assuming no correlation. However, as more correlation is provided, the materialized views provide comparable performance to, and can occasionally beat, the column stores.

With the cost of storage essentially free, materialized views can easily be included with a row store, which can help the relative performance of the row store. The materialized views will also be affected by the selectivity of the query, in proportion to the number of columns in the view. We did not study additional projections or sort orders for the column store since, in this case, they will most likely not provide a performance benefit.

### 3.6.9 Joins

Finally, we also examined joins with different selectivities. These experiments used a hybrid hash join [57] with enough pages allocated to ensure the inner table fits in memory. The two tables used in the join have either two columns or four columns accessed. The SQL for the two-column case would be:

```sql
SELECT temp1.column2, temp2.column5
FROM temp1, temp2
WHERE temp1.column9 = temp2.column9 AND temp1.column2 \geq x
AND temp2.column5 \geq y;
```

Temp1 and Temp2 are both table Narrow-E for the row store. For the column store, Temp1 is column 2 and column 9 from Narrow-E; Temp2 is column 5 and column 9. For the four-column case, table Temp1 also includes columns 1 and 3, and table Temp2 also includes columns 7 and 10. Column 9 was chosen as the join attribute since each value occurs a small number of times within the table, and the number of tuples passing a join increases as a square of the number of duplicate matching join attribute values. The other columns were chosen at random so there would be no overlap in Temp1 and Temp2, besides column 9.
Figure 3.9 Materialized views for the Narrow-E table. The top figure presents the results for two materialized views with five columns in each view. The bottom figure presents the results for five materialized views with two columns each.
We chose to use either two or four columns from the table for a variety of reasons. At five or more columns, the likelihood of running out of memory increases. Additionally, row stores outperform column stores as more columns are accessed. On the other hand, consistently using just the join attribute for both tables was unlikely in real workloads. Hence, we chose two and four columns to get two sets of results without severely impacting the column store’s performance.

The variables x and y in the query are varied to return either 10% or 100% of the rows for the table, and the query is performed four times to get times for scan selectivity factors of 100/100, 100/10, 10/100 and 10/10. The number of resulting tuples is 234M, 22M, 28M and 2.6M, respectively.

The inner table is the one with the fewest rows that pass the predicate. This table is scanned first, and its query is generated in the same way as the queries used for the results in the previous sections. However, once the tuple has been materialized, instead of being discarded, the join attribute is hashed, partitioned and inserted into the appropriate bucket page. After the first scan completes, the second scan begins, and probes for joins after the tuple has been materialized. If the bucket is wholly in memory, the resulting join tuple(s) is (are) materialized. If the bucket is not in memory, the tuple is written to a page and is processed after the scan is complete. This plan uses an early materialization strategy, as per the results of [9].

Figure 3.10-top presents the elapsed time for the join for both row stores and column stores for the two column tables, with the given scan selectivities. Figure 3.10-bottom presents the results for the four column tables, but does not include the 100/100 case in the results because the inner table does not fit in memory. Each bar presents the elapsed time for the join, and the part of that time it takes to just perform the two scans. The scan results are as expected, and the join component of the time is always roughly equivalent between the column store and row store. Thus, the paradigm with the smaller scan time will also have the smaller join time, and the join time is greatly affected by the number of joined tuples that must be materialized.

3.7 Discussion

To begin the discussion, let us summarize the findings:
Figure 3.10 The top graph presents results for joining two two-column tables in a column store and row store. The x-axis labels give the storage paradigm and the selectivity of each scan. The bottom graph presents the results for joining two four-column tables.
• Read-optimized row stores and column stores compress to within a few bits of each other.

• Regardless of the table size or column types, the selectivity of the predicate can substantially change the relative performance of row stores and column stores.

• Row stores perform better compared to column stores when the rows are narrow.

• For a given selectivity, adding predicates also increases column store runtimes.

• Having more qualifying rows increases column store runtime.

• Materialized views, which are essentially a hybrid of row and column stores, can out-perform both of them, depending on the circumstances, but normally have performance somewhere between the two paradigms.

• Hybrid hash joins with early materialization do not change the relative performance of row stores and column stores.

While [28] reached some of the same conclusions, our results further quantify these findings and the extent to which they hold, and add some new results.

A general rule of thumb is that a column store will outperform a row store when I/O is the dominating factor in determining the response time of a query, but a row store can outperform a column store when processing time is the dominating constraint. The key point to note, as our results show, is that I/O becomes less of a factor for row stores with compression, and adding predicates, decreasing selectivities and referencing more columns makes CPU time more of a factor for column stores.

Row stores on slotted pages will most likely never beat column stores for read-optimized workloads since their bandwidth requirements are much higher than for even the uncompressed bit-aligned tables. However, a read-optimized row store can clearly outperform a column store under some conditions.

Row store designers must seriously reconsider two points: compression and schema design. Using aggressive compression techniques is critical to reducing the overall scan time for a row
store. In addition, our results along with those in [28] clearly demonstrate that, for the current generation of CPUs and disk drives, 20 bytes is a good average tuple size to aim for.

Additionally, some column store proponents have argued that, since column stores compress so much better than row stores, storing the data with multiple projections and sort orders is feasible and can provide even better speedups [58]. However, we have found that columns do not actually compress any better than read-optimized rows that employ bit compression and delta encoding. Since it is now feasible to store row stores in multiple projections and sort orders without a substantial storage overhead, developing techniques for selecting the best materialized views (keeping in mind the 20 bytes per view per row target) might prove to be beneficial, especially as our results from Section 3.6.8 show.

We have shown multiple ways to decrease the I/O requirements of a query workload. If the workload has many low-selectivity queries, or multiple predicates per query, the tuples could be even larger and still provide roughly the same performance as column stores. However, for workloads comprised of high selectivity queries that randomly select just one or two columns from a wide table that cannot be vertically partitioned, row stores cannot compete.
Chapter 4

DD: Trying to Solve Flash’s Random Write Problem

In the previous chapters, we studied compression, particularly as a way to balance processor and disk resources to achieve better overall performance. We now turn our focus to solid-state disks. The past few years have seen dramatic breakthroughs in SSDs: the prices have dropped while capacities have increased. Additionally, solid state disks provide much better random read behavior than standard HDDs, while using substantially less energy. However, some SSDs have much worse random write characteristics than hard disk drives (HDDs). This chapter introduces Dual-Drive (DD), a novel method to improve response time in SSDs with random-write intensive workloads, by using an additional disk that is solely dedicated to buffering random writes. DD can decrease response time by 20-30% in random-write intensive workloads without increasing the response time for other workload mixes.

4.1 Introduction

In the past few years, SSDs have garnered increased attention. Their capacities have increased to 32GB and above at prices below $1000. Solid-state disks have much better random read times than traditional disks and consume less power. Both these advantages arise because SSDs do not have moving parts: they are purely electronic. Unfortunately, some current SSDs have much worse random write characteristics compared to HDDs because an entire block (often 4 KB - 256 KB) must be erased before it can be overwritten.

Thus, while it currently makes sense to replace HDDs with SSDs in read-heavy workloads [40], they might not produce a performance advantage if the workload has many random writes. Because
poor random write behavior is intrinsic to the medium and NAND SSDs are accessed using the same standard SATA interface as HDDs, users must look to additional hardware and software to try to mitigate the problem.

Some in the OS community have looked to log-structured file systems as a solution [54]. In log-structured file systems, all writes are appended to the end of the log until a separate cleansing stage occurs. However, most operating systems’s default file system still updates data in place.

This chapter introduces Dual Drive (DD), a new technique to try to solve flash drives’s random write problem: we still want the primary copy of the data to be on the SSD. This technique uses a traditional HDD in tandem with a SSD. Random writes are written sequentially to an append-only buffer on the HDD, and structures are maintained in memory to map pages from the SSD to the HDD. A later cleansing step then moves the new data back to the SSD.

This technique can be viewed as a complement to the idea of HDDs being augmented with DRAM to improve read behavior. One could argue that DRAM could also be added to a SSD to cache writes, but there are two main problems with that idea:

1. DRAM, even battery-backed DRAM, is less persistent than a HDD in case of failure, so writes could still be lost.

2. The best performance for SSDs occurs when writes are coalesced, turning multiple random writes into one long sequential write. In general, it will take many random writes to get to the point where coalescing will be effective, which would require gigabytes of NV-DRAM.

On the other hand, HDDs are stable, large and cheap.

DD provides up to 20-30% improvement in total elapsed time for a random-write intensive workload. Unfortunately, DD does not provide substantial improvement for other workloads. This implementation of DD uses a low-end 32 GB SSD, a middle-line HDD and a small amount of memory. Some experiments were also performed with a higher-end SSD; in this case, DD did not provide a performance advantage.
As motivation, the read and write bandwidth for a low-end and higher-end SSD and a conventional disk are presented. Then we provide more details of DD’s implementation. Next, experimental methods are discussed, followed by results. The chapter finishes with related work and conclusions.

### 4.2 Observations

Bandwidth is one of the most important characteristics of a disk. Since SSDs are still relatively new, and still have widely varying characteristics, we decided to compare the sequential and random read and write bandwidth of two SSDs with that of a HDD.

The system used for all experiments in this chapter has three drives: an MTRON MSP-SATA 7000 series 32 GB SSD, a 32 GB Intel X25-E Extreme SSDSA2SH032G1C5, a 320GB 7200 RPM SATA Western Digital WD3200AAKS HDD, and another HDD, on which the operating system Red Hat Enterprise Linux 5 is run. The processor is an Intel Core 2 Duo, and the system has 2 GB RAM.

For this experiment, both drives were accessed as raw devices. One gigabyte was read from or written to the device using varying block sizes while the block size was varied from 2 KB to 1 MB (1024 KB). Thus, to read or write 1 GB using 2 KB blocks requires 512K accesses, while 1024 KB blocks requires 1024 accesses. For the “random” experiments, the accesses were performed at random locations within the 1 GB file. The results obtained for the read and write experiments are presented in Figure 4.1, respectively. The results for block sizes less than 16KB are not presented in those cases where the bandwidth is substantially less than 5MB/s due to their long run-times.

Figure 4.1-top shows that sequential read bandwidth for the three drives is 70-90 MB/s, regardless of block size. The SSDs also have much better random read performance than the conventional disk, though the bandwidth obtained varies depending on the block size and peaks at 128 KB and 1024 KB. For the HDD, the bandwidth increases proportionally to the block size.

Figure 4.1-bottom paints a dramatically different picture for writes. Now, even the sequential writes are affected by the block size, with a noticeable increase in bandwidth as the block sizes double from 2 KB to 4 KB and up to 32 KB. With a block size of 64 KB, the conventional HDD’s
bandwidth has leveled off to a steady state of 80 MB/s. The SSDs’s sequential write bandwidth appears more stable at block sizes of 64 KB and above, but further increases in block size improve performance to 55 MB/s for the MTRON SSD and 90 MB/s for the Intel SSD. The conventional disk has substantially better write performance than the MTRON SSD, while the Intel SSD has symmetric random and sequential write bandwidths that are comparable to the conventional drive’s sequential write bandwidth.

The SSDs appears to have three sweet spots for performing random reads, at block sizes 4 KB, 128 KB and 1 MB. These results are repeatable on our disk, as these figures are averages of ten runs. We are not sure why these sweet spots occur, but we believe it has to do with the physical design of the drive. The disk logically treats data in 1 MB units, while it has 64 2 KB pages per physical block. If the disk were to always return one physical block, 128 KB, it would explain the increase in bandwidth from 8 KB to 128 KB. However, that explanation does not explain the behavior at block sizes of 4 KB, 256 KB and 512 KB.

This difference between the conventional disk and the SSD is even more noticeable for random writes. For block sizes up to 128 KB, the conventional disk has roughly twice as much bandwidth as the MTRON SSD. This difference is due to the overheads related to each access. On average, each random write to the conventional disk has a 4 ms overhead, while the MTRON SSD has an 8 ms/access overhead. Some of this overhead may be due to the physical implementation of the SSD, but others have also observed the poor random write behavior of SSDs and devised various schemes to try to combat it (see Related Work). However, the Intel SSD has the same sequential and random write bandwidths, and both are comparable to the conventional disk’s sequential write bandwidth. The Intel SSD is a higher-end disk; we have two hypotheses for its symmetric write bandwidths: the drive may be over-provisioned with flash blocks and is thus able to always hide write latencies, or the drive has a large, high-performance write cache.

4.3 DD: Dual Drives

The intuition behind DD is to try to avoid random writes to a database stored on a SSD. Instead of updating randomly written blocks in place on the SSD, DD appends such blocks to the HDD, and
Figure 4.1 Random and sequential read bandwidth (top) and write bandwidth (bottom) for a conventional disk (Conv) and a SSD (SSD).
then moves the data from the HDD to the SSD periodically during cleansing process. Thus, a block can either be in its expected location on the SSD or somewhere on the HDD. Our design employs two data structures to track the location of a block: SSDBlock-To-HDDPageNum, which maps an SSD block number to the block’s position on the HDD, and a reverse map of this information HDDPageNum-To-SSDBlock, to be used during the cleansing stage. On a read, the SSDBlock-To-HDDPageNum structure is checked; if no entry for the block is found, the block is read from the SSD, else it is read from the HDD at the location stored in the SSDBlock-To-HDDPageNum table. Figure 4.2 presents an overview of the layout and a simple example.

### 4.3.1 Algorithm

The algorithm that DD uses for reading and writing blocks is presented below. Random accesses are performed on a block-by-block basis, while sequential accesses that consist of multiple blocks can be grouped together. Since SSDs have fairly good sequential write performance, all sequential writes are directly sent to the SSD. Blocks are only moved from the HDD to the SSD during cleansing.

The algorithm breaks accesses down into three cases:

1. **Reads.** For each block of a read, the SSDBlock-To-HDDPageNum structure is checked. If an entry for the block exists, the block is read from the appropriate location on the HDD, otherwise the block is read from the SSD.

2. **Sequential Writes.** Each block of the sequential write is written directly to the SSD, since the SSD has reasonable sequential write performance. If the block had been on the HDD, the entries for the block are deleted from the SSDBlock-To-HDDPageNum and HDDPageNum-To-SSDBlock structures.

3. **Random Writes.** On a random write, if there is no room left in the buffer on the HDD, the cleansing procedure is performed.

In this example, the SSD’s block 2 was randomly written to the HDD’s block 3. There were then three random writes to blocks that have since been overwritten (obvious because they do not
Figure 4.2 Illustration of DD’s structures.
have entries in HDDPageNum-To-SSDBlock). Most recently, the SSD’s block 0 was written to block 7 on the HDD.

Then the block is appended to the HDD and the SSDBlock-To-HDDPageNum and HDDPageNum-To-SSDBlock structures are updated.

The idea behind the cleansing procedure is very similar to periodic garbage collection. The cleansing process is simply a two-phase external merge sort, where the valid blocks from the HDDPageNum-To-SSDBlock structure are read into memory and sorted based on the SSD block number. Instead of writing the data back to the HDD during the final merge phase, the blocks are written to the SSD.

In the example in Figure 4.2, there have been eight random writes. However, all but two of these writes were subsequently overwritten (the writes to SSD blocks 0 and 2). The next random write would fill the HDD, so the random write after that would then trigger a cleansing.

4.3.2 Failure Recovery

Though the HDD will not lose data during a power failure, the information stored in HDDPageNum-To-SSDBlock and SSDBlock-To-HDDPageNum will be lost. To achieve durability, the SSD position of the block is written, along with the block, to the HDD. On a sequential write, entries are removed from the SSDBlock-To-HDDPageNum and HDDPageNum-To-SSDBlock mappings; this event should also be logged.

In the event of a failure, the HDD is sequentially scanned. The first log record is read. This record gives which solid-state block follows at hddPageNum. The SSDBlock-To-HDDPageNum and HDDPageNum-To-SSDBlock mappings are reconstructed with these <ssd block, hddPageNum> pairs, and the HDD seeks to the next log record (32 KB away). Either this record provides another solid-state block number, or it indicates that an entry must be removed from the mappings. If the entry does not indicate removal, the previous procedure applies. Otherwise, the mappings' entries for the given SSD block are deleted. Then the next log record is read.

Failure can also occur during cleansing. The easiest solution for coping with failure during cleansing is to use no more than half of the disk drive. Then, the sorted chunks are written to
the second half of the disk during the first phase of the merge sort. If failure occurs at any point during cleansing, the entire procedure is restarted once everything is back online. More space- and time-efficient failure recovery methods are possible, but they are more complex.

4.4 Experimental Setup

The goal for the experiment was to compare how DD performs versus a HDD or a SSD alone under a variety of possible workloads. These experiments do not include DD’s failure recovery mechanisms. Both the SSD and HDD are accessed as raw devices. The drives are accessed a block at a time, where each block is 32 KB and the file size was set to 16 GB. A block size of 32 KB was chosen since it represents a sweet spot for the two devices – big enough to start getting reasonable performance without being too big.

The experiment has three knobs: read/ write, random/ sequential, hotcold/ uniform. The read/ write knob determines the percentage of accesses that are reads or writes. The random/ sequential knob determines the percentage of accesses that are random or sequential; sequential accesses touch 1-128 blocks a time, while random accesses only touch one block. The hotcold/ uniform knob is really a switch: either each block is equally likely to be accessed (uniform), or a small set of blocks are highly likely to be accessed (hotcold).

On each new access, two random numbers are drawn. One determines if that access will be sequential or random, and the other determines if the action will be a read or a write. If the access is sequential, another random number is drawn to determine how many blocks will be accessed, from 1-128, uniformly distributed.

If the workload is uniform, each block is equally likely to be accessed. However, if the workload is hotcold, 90% of the accesses will be to a block in the first 10% of the file, with the other 10% of the accesses going to the other 90% of the file. Within a hot or cold region, each block is equally likely to be chosen.

We wanted to test DD with a wide range of possible workloads, so the three device setups are each tested with four different synthetic read/ write and sequential/ random mixes. Additionally, each of these mixes is performed with both a hotcold workload and a uniform workload, for a total
of eight experiments that represent a wide range of possible workloads. The four mixes are:

<table>
<thead>
<tr>
<th></th>
<th>20% Sequential</th>
<th>80% Sequential</th>
</tr>
</thead>
<tbody>
<tr>
<td>20% Reads</td>
<td>M1: random writes</td>
<td>M2: sequential writes</td>
</tr>
<tr>
<td>80% Reads</td>
<td>M3: random reads</td>
<td>M4: sequential reads</td>
</tr>
</tbody>
</table>

The experiments are stopped immediately after a cleansing cycle has occurred, since this is when DD would have the worst performance relative to the other disks. The question then becomes how often the cleansing should occur and how large the buffer on the HDD should be.

The random write experiments are run until, on average, every block of the 16 GB file on the SSD has been randomly written once, and one block has been written a second time, thus forcing a cleansing of the HDD. The sensitivity of the random-write intensive mixes (M1) to the size of the buffer on the HDD space is studied in the next section.

Mixes M2-M4 terminate after 32K, 128K and 8K random writes have been performed, respectively. Thus, they use 1GB, 4GB, and 256MB, respectively, on the HDD.

### 4.5 Results

Four sets of results are presented: those for write-intensive workloads, those for read-intensive workloads, a sensitivity analysis of the cleansing frequency, and comparison of DD to using a log-structured file system.

#### 4.5.1 Write-Intensive Workloads

DD was designed to improve the performance of random writes to a database stored on a SSD by incorporating an additional HDD. Figure 4.3 presents elapsed times for DD, a HDD alone, a MTRON SSD alone, and an Intel SSD alone. The DD times are broken down into two portions: the main algorithm, and the cleansing stage. The write-intensive workloads are mixes M1 and M2 (top and bottom), for both the hotcold and uniform workload.
For the hotcold workload with 20% sequential accesses, DD with the MTRON SSD provides roughly a 25-30% performance improvement over both the HDD and SSD alone. The same mix with the uniform workload again outperforms the HDD and MTRON alone, but the performance improvement is closer to 6%. This difference in performance improvement is partly due to the increased cleansing costs with the uniform workload. The hotcold workload writes to far fewer blocks than the uniform workload, so cleansing takes correspondingly less time, and more blocks are coalesced.

The elapsed times for the MTRON and DD hotcold 80% sequential mix are roughly the same, and outperform the HDD-only case. However, the same mix with the uniform workload shows the solid state drive performing noticeably worse than in the hotcold case, and DD is now worse than the HDD-only case. This difference arises because the sequential hotcold accesses are more likely to be sequential from one set of accesses to the other, since the hotcold region is 1.6GB, compared to the uniform workload’s 16GB region.

The experiments with the Intel SSD were less encouraging for DD: DD always had worse performance than the Intel SSD alone. This result is not surprising since the Intel SSD has such has overall bandwidths and is particularly good at random writes, which DD was designed to overcome. In all of our experiments, the Intel SSD had elapsed times that were no more than half the elapsed times of the HDD only experiments.

### 4.5.2 Read-Intensive Workloads

Though DD was designed for write-intensive workloads, ideally it will never have a longer elapsed time than the SSD alone. Figure 4.4 presents the elapsed times for mixes M3 and M4 for the two workloads. Unfortunately, in all four mixes, the SSD alone slightly outperforms DD. However, in three of the four experiments, the difference is quite small – roughly 1-2%.

DD and the MTRON SSD noticeably outperform the HDD alone in the hotcold workload. However, the SSD again performs 10% worse in the uniform, 80% sequential case than in the corresponding hotcold case.
Figure 4.3 Write-intensive workload (80% writes).

Figure 4.4 Read-intensive workload (80% writes).
4.5.3 Sensitivity

In order to achieve the best performance, it is important to clean at the proper interval. If cleansing occurs too soon or frequently, the writes back to the SSD will still essentially be random, thus negating any benefits DD provides. However, if cleansing is performed too infrequently, more random reads will be directed to the HDD instead of the SSD.

Figure 4.5 presents the elapsed times for the random-write intensive mix M1 for the hotcold and uniform workloads at various HDD buffer sizes and with the MTRON drive. Five buffer sizes are presented for each workload; the total number of reads and writes are the same in each test, but the 1GB buffer has 16 cleansings compared to the 16GB buffer, which only has one. For the hotcold workloads, the 1GB buffer still results in a lower elapsed time than the SSD (from Figure 4.3); however, in the uniform case, the 1GB buffer takes noticeably longer than both the HDD and the SSD. For both workloads, for the cleansing frequencies that were studied, the bigger the HDD buffer, the shorter the elapsed time.

Since the uniform random-write intensive workload had substantial benefits from increased cleansing intervals, more experiments were performed. At 16GB (shown), the uniform workload’s elapsed time is 6% shorter than that of the SSD alone. However, at 32GB, the improvement increases to 11%, and, at 64GB, the improvement is 16% (not shown). Buffer sizes above 64 GB have not been studied. This workload benefits from larger buffer sizes because the region of the file that is being written to is roughly ten times larger than the hot region in the hotcold mix. For instance, there are 512K 32 KB blocks in the 16 GB file. After 512K random writes, 55,262 unique blocks have been randomly written in the hotcold workload, while 327,815 have been written in the uniform workload. Thus, either all or almost all of the blocks in the hotcold region have been randomly written, so many of those writes will be coalesced. However, only 63% of the uniform region has been randomly written; thus coalescing is not very effective.

4.5.4 Comparison to NILFS

Log-structured file systems were created to help improve write throughput by replacing in-place updates with appends to the head of a log [54]. Since log-structured file systems minimize
random writes, they would appear ideal for SSDs. Hence, the experiments were also performed with the database on the MTRON SSD while running NILFS, a log-structured file system [36].

Because log-structured file systems do not perform in-place updates, they are very sensitive to the size of the disk and the amount of write traffic. Our write-intensive workloads have too much write traffic for NILFS’s garbage collection to keep up; the experiments abort due to insufficient disk space. Instead, only the read-intensive mixes were run, and those were scaled down by a factor of 4. So instead of reading and writing to a 16 GB logical area, as in the previous experiments, a 4GB NILFS file is created and accessed.

Figure 4.6 shows that NILFS has a shorter elapsed time than DD or the raw MTRON SSD for the read-intensive workloads. However, the garbage collector was unable to keep up with the volume of random writes being performed. Thus, while only 4GB file was written to, at the end of the run, the log was much larger. Had the costs of garbage collection been included in the elapsed times for NILFS, NILFS would most likely not have provided a performance improvement. Because NILFS is not yet production-ready and the developers recognize that garbage collection needs to be improved, it would be interesting to rerun the experiments after it, or another log-structured file system, matures.

4.6 Related Work

Multiple researchers have also studied adding more RAM to a SSD to improve random write performance. BPLRU is a new replacement policy for RAM that might be in the SSD (but above the FTL) as a buffer to cache writes [33]. They found a 44% performance improvement on a trace-driven simulation of a MS Office 2003 installation. Chameleon uses FRAM to store small random writes, especially those to metadata in the FTL and claims to provide 21% performance improvement [64]. And Birrell, et al. suggest using a large volatile RAM plus extra data structures to map disk logical blocks to flash addresses in order to achieve better random write performance [14].

New FTL algorithms have been studied extensively in the literature [18]. Kim, et al. introduce a log block scheme in the FTL to remap writes [34]. These log blocks are used as temporary storage for overwrites: instead of writing the data in place, the sector is forwarded to its appropriate log
Figure 4.5  Sensitivity of the workloads to the buffer size for the write-intensive mix (case 1), with the MTRON drive.

Figure 4.6  Comparison of DD, the raw MTRON SSD and the MTRON SSD with NILFS, a log-structured file system, for the read-intensive mixes (cases 3 and 4).
block. Each log block is direct-mapped, so if a given sector’s log block is in use, it cannot use another (free) block in the log. [40] improves on this scheme by making the log blocks fully associative, resulting in better space utilization and thus lower elapsed time and fewer erasures.

Log-structured file systems were introduced in [54] to improve random write performance. The basic idea is that, instead of updating files in place, the updates should be appended to the end of a log, then, at some point, the dead blocks are cleaned. At least one journaling file system has been built specifically for flash devices, but its use is more focused toward embedded devices [7]. Myers’s thesis found NILFS greatly improved random write performance but at the cost of random reads [44].

4.7 Conclusions

In conclusion, this chapter introduces a new approach for improving random write performance for databases stored on low-end SSDs. DD combines a SSD with a HDD and in-memory data structures to map random writes to the HDD. The technique can provide up to a 30% performance improvement in random-write intensive workloads, without hurting the performance of other workloads compared to a low-end SSD alone.
Chapter 5

Extending the Buffer Pool with a Solid State Disk

We now continue our exploration of novel uses for a solid-state disk (SSD) by using a SSD as a secondary buffer pool, called the SSD-BM.

5.1 Introduction

In the past few years, SSDs have changed from niche products to mainstream devices as their prices have dropped and capacities have risen. These drives are entirely digital: instead of spinning platters, as in a traditional disk drive (HDDs), they use flash chips. Traditional disks have poor random read bandwidth, since the head must be moved to the correct cylinder, then the platter must spin to the correct sector. However, since SSDs have no moving parts, there is almost no seek time to move from one page on the drive to another, and thus they have good random read performance.

SSDs and HDDs have comparable sequential read and write latencies and bandwidths, since the seek times are dominated by the time to transfer the block. Hard disks have poor random write bandwidth for the same reason they have poor random read performance.

However, SSDs can also have poor random write behavior. This occurs because flash chips are cleared in large blocks – often about 128 KB in size. Since each bit in the block can only be written once; the second write to a bit requires the entire block be erased first. To improve performance, the SSD erases blocks in the background, but if there are no free blocks, the erase time cannot be masked.
Though SSDs are dropping in price, they are still several times more expensive than HDD of the same size. For instance, the cheapest 256 GB SSD available from newegg.com costs $579, while the cheapest HDD of roughly the same size costs $48 – a factor of twelve less. Because of the price and performance discrepancies, one question that arises is where in the DBMS storage hierarchy the SSD should lie. An SSD should definitely be lower in the hierarchy than memory (RAM), since memory maintains a low access latency and a high bandwidth, but at a relatively steep monetary cost. It is tempting to treat SSDs as if they were a larger, slower RAM, but while RAM has symmetric read and write behavior, SSDs can exhibit poor random write performance.

This chapter proposes a solid-state disk buffer manager (SSD-BM). In this design, a SSD is used as a secondary database buffer pool, with the key innovation that pages are only written sequentially to the SSD when they are ejected from the buffer pool in RAM. Thus, we can take advantage of a SSD’s better random read performance without having to incur the penalty of slow random writes.

Ideally, incorporating SSD-BM into a relational DBMS has the potential to provide performance better than using the HDD alone. However, the performance of SSDs varies among the different products. A high-end, enterprise-class SSD might provide equivalent random and sequential write performance, while there might be substantial differences in bandwidth with a commodity SSD. Of course, enterprise SSDs are significantly more expensive ($83 for a bottom-of-the line 32 GB SSD, vs $383 for a fast Intel 32 GB SSD), hence we will study performance of the SSD-BM design from several perspectives:

- We will compare a system with the SSD-BM to a system that only uses a HDD and to a system that only uses a SSD, for two different classes of SSDs.
- Further, we will vary the size of the SSD-BM, to see how the system performance, depending on the percentage of the database that can fit in SSD-BM.
- We will also vary the query workload.

The main contributions of this chapter are:
• The introduction of SSD-BM, with the key insight of only writing to the SSD sequentially.
• A working implementation of SSD-BM in MySQL using the InnoDB storage engine.
• A detailed performance evaluation with different replacement policies for SSD-BM, workloads and SSDs.

Since SSDs are a new technology, much of the published research (discussed in related work) uses SSDs for specific functions in a DBMS, attempts to improve their random write performance, or completely replaces HDDs with SSDs. To the authors’ knowledge, there is no other published research that uses an SSD as a secondary DBMS buffer pool.

In the rest of the chapter, SSD-BM is presented in more detail, followed by a description of the system and experimental set up. Next, performance results are provided. This chapter concludes with a discussion of related and future work and the main results of the research.

5.2 SSD-BM Implementation

This section begins with a discussion of how the primary buffer pool of the InnoDB storage manager works. The next subsection discusses how SSD-BM and the primary buffer pool interact. Finally, multiple replacement policies for the SSD-BM are discussed.

5.2.1 Primary Buffer Pool

SSD-BM was implemented in MySQL using a modified InnoDB storage engine [6]. MySQL has a clean separation between its query engine and the storage engine, which made it attractive for doing storage research. Additionally, InnoDB provides transactional semantics, commit, rollback and recovery, and stores data on pages.

The InnoDB buffer pool stores pages of tables and their associated metadata in buffer frames in memory. InnoDB maintains three important lists for the buffer pool: the free list; the LRU list, which lists un-pinned pages in near-LRU order; and the flush list, which contains the control blocks for pages that are dirty in the buffer pool and have not yet been written to disk. The buffer pool uses an LRU replacement policy by default.
5.2.1.1 Page Reads

On a page read, the buffer pool is checked. If that page is in the buffer pool, a pointer to it is returned. Otherwise, a buffer frame must be found, either from the free list (preferably) or from the end of the LRU list. The frame is then fixed in memory, its latch is reserved, and a read I/O request is queued. The latch is freed after the I/O completes. The read block is then moved to the front of the LRU list. InnoDB tracks consecutive read requests, and prefetches pages with a strided pattern or where multiple pages from the same area of the file are being used.

5.2.1.2 Page Writes and Flushes

If a buffer block was modified during a transaction, the block is inserted into the flush list at commit.

In order to improve performance, instead of flushing dirty pages when the page is evicted from the buffer pool, InnoDB tries to flush pages in the background. Additionally, InnoDB writes every flushed page twice: once into a double-write buffer, and then to its location on disk. The double-write buffer is a performance optimization designed to reduce the number of fsyncs. Instead of having to fsync after each individual page has been flushed (to ensure each page was fully written), the double-write buffer batches updates. The double-write buffer is written sequentially to a special area on disk and an fsync is performed. Then the pages are updated in-place on disk. After all pages have been updated, another fsync is performed. If a failure occurs after part of one page has been written to disk, on recovery, the full, consistent page(s) is copied from the double-write buffer.

When a page is read into the buffer pool, a new frame is requested, or the buffer pool looks for a free block, InnoDB checks to see if there are enough free frames (determined by a user-defined threshold). If there are not enough, InnoDB searches the LRU list for old pages that are no longer pinned in the buffer pool. If any of these pages are dirty, they are moved into the in-memory double-write buffer. Further, if there are any other dirty pages that are stored in the file close to these pages, they are also moved to the double-write buffer. Once the in-memory double-write buffer is full, the pages are flushed, as was just described. Flushes are performed asynchronously, and once the page has been written to disk, it is removed from the flush list.
5.2.2 SSD-BM

For now, the SSD-BM can be thought of as a circular queue stored on an SSD that is used to cache pages that have been evicted from the buffer pool. Pages are only added to the end of the queue (or wrapped around), and nodes of the queue can be randomly accessed or invalidated. Since SSD-BM acts a secondary buffer pool, there are four main events in the primary buffer pool that must be monitored: a miss to the primary buffer pool, a page updated in the primary buffer pool, a page evicted from the buffer pool, and a dirty page flushed from the buffer pool. This subsection first discusses the structures used by the SSD-BM. Then, for each of the four events, the desired behavior in the SSD-BM and how it is implemented are discussed. In the next subsection, the SSD-BM replacement policy is discussed.

Since most SSDs have a steep random write penalty, SSD-BM attempts to avoid random writes. This innovation means SSD-BM must employ a different implementation strategy than the primary buffer pool. For instance, InnoDB keeps a free list, an LRU list and a flush list to track the state of the buffer frames. The flush list is an optimization to perform writes in the background, but the other two lists are necessary in order to track where a new page can be inserted and which page should be evicted. In the SSD-BM, those two lists are not necessary, since it is managed as a queue. The tail of the queue will be referred to as curFrame.

Since a page can be anywhere in the SSD-BM, which pages are stored in the SSD-BM and their location must be maintained. This metadata is stored in two in-memory hash tables: one that hashes page identifiers to their location in the SSD-BM (called the PageNum-to-SSDFrame hash table), and another that hashes SSD-BM locations to page number (SSDFrame-to-PageNum hash table). At a minimum, both hash tables store the (disk page, SSD-BM frame number) mapping and whether the page is dirty. The PageNum-to-SSDFrame hash table also stores the number of times the page has been read from the SSD-BM into the primary buffer pool. This information is used by the replacement policy.

The combination of the curFrame, SSD Frame-to-PageNum hash table and PageNum-to-SSD Frame hash table are necessary for the SSD-BM to perform its functions. The desired SSD-BM
behavior and its implementation for each of the four events mentioned at the beginning of this subsection will now be discussed.

5.2.2.1 Primary Miss

On a miss to the primary buffer pool, if the page is in the SSD-BM, it is returned from it, and the number of uses for that page is incremented. Otherwise, the page is read from the secondary storage directly into the primary buffer pool. The desired sequence of events for both an SSD-BM hit (a) and miss (b) are illustrated in Figure 5.1.

![Figure 5.1 Illustration of what happens in the SSD-BM when there is a miss in the primary buffer pool.](image)

5.2.2.2 Primary Update

Pages must be kept consistent between the primary buffer pool and the SSD-BM. Hence, when a page from the primary buffer pool is updated, if that page exists in the SSD-BM, it should be invalidated (rather than immediately written through to the SSD-BM). It does not make sense to write the page back, since, if it is clean, the page is already on disk, and, if it is dirty, it is no longer the most recent version. Figure 5.2 presents a PageNum-to-SSDFrame hash table entry being invalidated due to a page update in the buffer pool.
5.2.2.3 Primary Eviction

When a page (either dirty or clean) is evicted from the buffer pool (the page is moved to the free list), instead of being discarded (if the page is clean) or written back to disk (if it is still dirty), it is inserted into the SSD-BM if it is not already there. Figure 5.3 shows what happens when a page is evicted from the buffer pool that is not already in the SSD-BM. Again, the SSDFrame-to-PageNum hash table is not shown.

5.2.2.4 Page Flush

Recall that in InnoDB, dirty pages are not flushed when the page is evicted from the buffer pool. Instead, they are flushed before eviction, in blocks of 100 (the double-write buffer size). When the SSD-BM is present, the pages in the double-write buffer are still first written to the secondary storage, since the write is sequential. Normally (without the SSD-BM), after the double-write buffer has been written, the pages are written to their proper locations on disk. However, with the SSD-BM, the pages are instead written to this secondary buffer pool. These pages are not written back to the secondary storage until they are evicted from the SSD-BM.
Implementation: When InnoDB would normally write a dirty page back to its location on disk, the write is instead directed to the SSD-BM. If the page is already in the PageNum-to-SSDFrame hash table, its SSDFrame-to-PageNum hash table entry should be deleted, and a new one created for the new location (since SSD-BM does not perform in-place updates). Additionally, the SSD-BM Location field in the PageNum-to-SSDFrame hash table should be updated, the page should be marked as Dirty, and the number of uses for the page set to zero.

If the page had not been in the SSD-BM previously, new entries are created in both hash tables. Figure 5.4 illustrates the expected behavior, omitting the SSDFrame-to-PageNum hash table.

5.2.2.5 Comments on SSDFrame-to-PageNum Hash Table

Only the PageNum-to-SSDFrame hash table must be searched while the SSD-BM is filling. However, the SSDFrame-to-PageNum hash table is useful on two occasions. First, once the SSD-BM is full, a primary buffer pool eviction might require a page to be evicted from the SSD-BM and its location rewritten. The SSDFrame-to-PageNum hash table makes it easy to determine whether a valid and dirty disk page is stored at that location. In the absence of the SSDFrame-to-PageNum hash table, the PageNum-to-SSDFrame hash table would have to be scanned (not hashed into) to find if there is a valid disk page for that location.
Figure 5.4 Illustration of what happens in the SSD-BM when a page is flushed from the primary buffer pool.

The SSDFrame-to-PageNum hash table is also accessed on database shutdown, to ensure that all dirty pages have been flushed to the HDD. The SSDFrame-to-PageNum hash table is scanned for each frame that contains a valid page. If the page is dirty, it is then written back to the HDD. Walking the PageNum-to-SSDFrame hash table in ascending order might result in better shutdown performance, since fewer random writes might be incurred to the HDD. However, since database shutdown is the uncommon case, there is a greater focus on correctness than performance.

5.2.3 Replacement Policy

In general, writes to the SSD-BM are performed sequentially. However, hot pages should also be preserved in the SSD-BM. We have studied two different replacement policies that try to achieve both goals: the One Buffer and the Two Buffer approaches. In the One Buffer approach, the entire SSD-BM is treated more-or-less as a single circular queue, while the Two Buffer Pool approach uses two circular queues: one for new or cold data, and another for hot data.
5.2.3.1 One Buffer Policy

For both policies, as the SSD-BM fills, pages are written sequentially. However, once the SSD-BM is full, the frames must be overwritten. A really naive algorithm to handle replacement simply writes dirty pages directly to the HDD. The steps would be:

1. Increment the SSD-BM location, curFrame. If the location is past the end of the SSD buffer pool, reset curFrame to zero.

2. Check the SSDFrame-to-PageNum hash table to see if there is a valid page stored at curFrame. If so, check whether that page is dirty. If dirty, it cannot simply be discarded as it is the only valid copy of the page. Hence, the page is read from the SSD into memory and then back to its proper location on the disk. Delete the PageNum-to-SSDFrame and SSDFrame-to-PageNum hash table entries for the page.

3. Insert the new page at curFrame and add hash entries.

This policy is easy to implement but can lead to poor performance, since it discards the page regardless of how hot it may be. Hence, it only fulfills one of the stated goals: that pages are written to the SSD-BM sequentially. An improved policy, called the One Buffer Policy, only overwrites a frame if that page had been used less than some threshold number of times. It still writes pages sequentially, but it tries to keep hot pages in the SSD-BM. This algorithm can be described as follows:

1. Increment curFrame. If the frame is past the end of the SSD buffer pool, reset curFrame to zero.

2. Check the SSDFrame-to-PageNum hash table to see if there is a valid page stored in the frame specified by curFrame. If not, skip to Step 4. Else, if the page to be replaced has been used more than a threshold number of times (in our experiments, we use the average number of uses of all pages in the SSD-BM), reset the counter on the page (# uses = 0) and go back to Step 1; otherwise, proceed to Step 3. To ensure that a new page will be inserted, we limit
the number of times we repeat Steps 1 and 2 to five. This number could be tuned if it hurts performance.

3. If Steps 1 and 2 have been performed five times, or the page at curFrame was used less than a threshold number of times, check to see if that page is dirty. If it is dirty, the page is read into memory and then written to its proper location on disk. Delete the PageNum-to-SSDFrame hash table and SSDFrame-to-PageNum hash table entries for the page.

4. Insert the new page at curFrame and update the hash tables.

The operation of the One Buffer policy is illustrated by Figure 5.5. In the figure, the unmodified page P8 needs to be added to the SSD-BM. curFrame starts at zero, and pages P1-P7 are already in the PageNum-to-SSDFrame hash table at the stated location. Dirty entries in the SSD-BM are shaded. The SSDFrame-to-PageNum hash table and primary buffer pool have been omitted for clarity, and the events are listed in order, along with what step of the algorithm they correspond to.

The policy begins by incrementing the value of curFrame from zero to one. Then, the SSD-Frame-to-PageNum hash table is checked to see if there is a valid page in that frame. By searching the PageNum-to-SSDFrame hash table by SSD-BM Frame number, it can be determined that P3 is located at SSD-BM[1]. The page is clean and has been used two times. In total, the pages in the SSD-BM have been used four times (2+1+1), and there are seven entries in the SSD-BM, so the average use is 4/7. Since two is greater than 4/7, P3 should not be ejected, and its number of uses should be reset (“-” -> 0”). Step 1 is then repeated, so curFrame is now two. P2 is at location 2; it is dirty and has not been used. Hence, P2 should be evicted from the SSD-BM, since its usage is lower than the threshold (now 2/7), and P8 should be written to its location. However, since P2 is dirty, it must first be written back to the HDD and its hash entries deleted (Step 3). The deletion is indicated by the lighter font color. Finally, P8 is written to frame number 2, and the hash entries are updated (bold italics).
5.2.3.2 Two Buffer Pool Policy

We have also created and studied a slightly more complex replacement algorithm, that logically partitions the SSD-BM into two buffer pools: one for hot pages, and one for cold pages. The most recently written page in the cold region is at curFrame1, and the most recently written page in the hot region is at curFrame2. By splitting the SSD-BM into two pieces, our goal is to ensure that hot pages remain in the buffer for as long as possible.

Initially, pages are inserted sequentially into the SSD-BM until it is full, just as with the One Buffer Policy. Every time the page is read back into the primary buffer pool, the usage counter is incremented. At first, the hot and cold regions are not distinguished, which allows the first set of pages to be in the buffer longer, increasing the probability of reuse.

Once every frame of the SSD-BM is occupied, when a page needs to be evicted from the cold region, its usage counter is checked. If the counter is above a threshold, instead of being discarded or written back to disk (if dirty), the page is promoted to the hot region. If the page selected for replacement in the hot region is valid, it is then written to the new tail of the cold region, and the page in that frame is either written back or discarded and its statistics are cleared. Finally, the new page is written to the cold region.

The steps are:
1. Increment curFrame1. If that frame is past the end of the cold region, reset curFrame1 to zero. (If this is the first pass through the SSD-BM, the regions should be ignored, and frames should just be sequentially written.)

2. Check the SSDFrame-to-PageNum hash table to see if there is a valid page stored at curFrame1.
   (a) If the page in curFrame1 is invalid, skip to Step 4.
   (b) If the page is valid but has been used less than a threshold number of times, check to see if it is dirty. If so, write it back to disk and delete the page’s hash table entries. Then go to Step 4.
   (c) Otherwise, the page is valid and hot, so move to Step 3, as this page should be promoted to the hot region.

3. This step promotes a hot page to the hot region, and might demote a hot page to the cold region.
   (a) Increment curFrame2. If curFrame2 passes the end of the hot region, reset its position to the beginning of the region.
   (b) Check the SSDFrame-to-PageNum hash table for an entry at SSDFrame2. If there is not one, insert the hot page into the buffer and add hash entries, then go to Step 4.
   (c) The page at curFrame2 is valid (and had been hot), so we do not want to totally evict it from the SSD-BM. Instead, the page gets demoted to the cold region.
      i. Perform Step 1. This new location will be referred to as curFrame1-2.
      ii. If the page at curFrame1-2 is valid and modified, write it back to disk.
      iii. Delete the hash entries for the page at curFrame1-2 (if the page is valid).
      iv. Write the page from curFrame2 into curFrame1-2, update the hash entries for the page, and clear the number of uses.
   (d) Insert the hot page into the SSD-BM at curFrame2, and update the hash table entries.
4. Insert the new page, at curFrame1 and add or update the hash entries.

This Two Buffer Pool policy is illustrated in Figure 5.6. These two diagrams represent the initial state, with writes and updates ordered, and the end state for inserting a new page. The PageNum-to-SSDFrame hash table has the same contents as in the one buffer example. SSD-BM is split into two equal-sized regions: the cold region and the hot region. curFrame1 is zero and curFrame2 is nine. In Step 1, curFrame1 is incremented to one. The page in SSD-BM[1] is P3. It is valid, clean, and has been used twice. For this example, the threshold for promotion (from the cold region to the hot region) is “# Uses > 0.” Thus, P3 should be promoted.

In Step 3, care must be taken not to overwrite or lose a valid page. Since curFrame2 had been nine, incrementing it rolls the value around to five (3a). The page at SSD-BM[5] is P6; it is valid, clean, and has been used once. Since it had been hot, we want to keep it in the buffer pool for a little longer, so it gets demoted to the cold region. The next curFrame1 location is 2. The page at SSD-BM[2] is dirty, so it must be written back to disk and its hash table entries deleted (Steps 3cii and iii). Now page P6 can be written to SSD-BM[2] (3civ) and P3 gets written to SSD-BM[5] (3d), and their hash table entries are updated. Finally, P8 gets written to SSD-BM[1].

The idea behind splitting the SSD-BM into two regions is to try to increase the probability that hot pages will remain in the SSD-BM. Further, the demotion to the cold region prolongs a page’s residence in the SSD-BM, and allows formerly hot pages to gracefully fall out of the SSD-BM.

![Figure 5.6 Before (left) and after (right) page P8 is added to the SSD-BM in the Two Buffer Pool replacement policy.](image)
5.2.4 Consistency

MySQL and the InnoDB storage engine are multithreaded. In order to ensure consistency and avoid race conditions, curFrame1 (and curFrame2) are only incremented while under the protection of a mutex. Further, each entry in the SSDFrame-to-PageNum hash table is protected by a mutex, and the PageNum-to-SSDFrame hash table is only updated and read while under the protection of the SSDFrame-to-PageNum hash table’s mutex. Otherwise, two threads could be attempting to write the same location in the SSD-BM, or one thread could try to read a page from the SSD-BM only to find it is no longer there.

5.2.5 Recovery

Log pages are always written directly to disk, bypassing the SSD-BM completely.

Crash recovery has not been implemented, however, it should be straightforward to modify InnoDB’s recovery procedure to work with SSD-BM since the log is still written to the HDD. At DB restart, InnoDB looks for a checkpoint label written to the log files. This checkpoint provides the largest LSN of what is expected on disk. The SSD-BM should be scanned for pages with an LSN less than the checkpoint’s LSN. If the page has an LSN less than the checkpointed LSN and larger than that page’s LSN that is stored on disk, the page should be copied from SSD-BM to the HDD, since the HDD has an old copy. This process can be slow, but it should return the database to a consistent state. Finally, InnoDB will scan the log files forward from the checkpoint, applying the changes stored in the logs to the database.

5.3 Experiments

The goal of the experiments we conducted was to understand how well the SSD-BM works as a secondary buffer cache and to determine when, if ever, the SSD-BM should be used. This section provides the configuration of the system tested: disk speeds, system set up, and workloads.
5.3.1 Disk Performance

This work is motivated by the fact that some SSDs have poor random write performance but all have good random read performance. To illustrate, we presented bandwidths for random and sequential reads and writes for three disks in the previous chapter in Figure 4.1. We study two different SSDs: an MTRON MSP-SATA 7000 series 32 GB SSD and a 32 GB Intel X25-E Extreme SSDSA2SH032G1C5. The HDD is a 320 GB 7200 RPM SATA Western Digital WD3200AAKS.

5.3.2 System

SSD-BM has been fully implemented in MySQL 5.0.67 using the InnoDB storage engine. The database and buffer pool use a 32KB page size, not the default 16 KB page size. We used a larger page size in order to achieve better random read and write performance from both the SSDs and the HDD, as can be seen from the bandwidths presented in Figure 4.1.

The experiments are performed on an Intel Core 2 Duo 2.4 GHz machine with 2 GB of RAM running RHEL 5; 1 GB of the RAM is allocated to the buffer pool. Both SSD-BM hash tables are limited to roughly 500,000 entries, which are stored in memory outside the buffer pool. In order to achieve better performance, the operating system, database and log files are run from a separate HDD.

5.3.3 Workload

In general, a DBMS will not benefit from a SSD unless the workload it is running has a significant number of random reads. Hence, we use a workload that combines a sequential-read-heavy thread with at least one thread doing repeated random reads and writes. These query threads consist of TPC-H queries over scale 10 Orders, Customers and Lineitem tables [22]. There are indexes on o_orderkey c_custkey and l_orderkey.

The sequential-read-heavy thread consists of TPC-H queries 1, 3, 4 and 6. Queries 1 and 6 only use Lineitem. Query 3 joins Customer, Orders and Lineitem, while Query 4 joins Lineitem and Orders. These queries were chosen since they have both scans and joins, and they had shorter run-times, which made the experiments tractable. One run of this query thread is comprised of
nine instances of each of the four queries, run back to back (e.g., Q1\textsubscript{1}, Q3\textsubscript{1}, Q4\textsubscript{1}, Q6\textsubscript{1}, Q1\textsubscript{2}, Q3\textsubscript{2}, Q4\textsubscript{2}, Q6\textsubscript{2}, etc).

The second stream consists of update queries. We used the TPC-H update transaction, which is normally only used to test ACID compliance. The stream updates Orders, Lineitem and a History table. The general update transaction is repeated up to 1,800,000 times, updating the first 25% of Orderkeys uniformly and at random.

Combined, these two threads mix short, random reads and writes with longer scans and joins and thus exhibit a range of behavior. However, we also wanted to test SSD-BM with other workloads: one with an additional update thread, and another where the data are skewed.

In the additional update thread, the updates are performed uniformly and at random across all Orderkeys. This thread adds two extra types of pressure to the workload: both more random reads and writes, and over a larger area (the entire Lineitem and Orders tables, rather than just a portion). Again, up to 1,800,000 updates are performed.

For both update threads, we say “up to” 1,800,000 updates are performed since the experiment is stopped when the query thread ends. This is done so that only steady-state behavior is measured. No updates that begin or end after the query thread finishes are counted.

The skewed data experiments are performed using generated [3] TPC-D tables [21]. TPC-D is based on TPC-H, but the data in the tables follow a Zipfian distribution rather than a uniform distribution; for our experiments, z=2.0. Further, we modified the updates to the TPC-D tables so that 80% of the updates are to 20% of the Orderkeys. The sequential-read-heavy thread remains the same.

5.4 Results

The performance analysis begins with a study of the One Buffer Policy for the two SSDs. We then studied buffer pool reuse statistics, and performed a sensitivity analysis for the Two Buffer Pool Policy. Finally, we examine performance results for the two different workloads: the additional update thread, and the skewed tables.
5.4.1 One Buffer Performance

For all of our experiments, the baseline performance is the system without the SSD-BM, and with the database tables either entirely on the SSD, or entirely on the HDD. For the experiments with the SSD-BM, we vary the SSD-BM’s size in order to emulate what would happen if we had different-sized SSDs (and thus what would happen when varying percentages of the database fit in the SSD-BM). We use 16 GB as a starting point, since that is how much space is required to hold the tables and indexes for the scale 10 Lineitem, Orders and Customer tables. We wanted to test on TPC-H data of a larger scale, such as scale 30, but our SSDs were only 32 GB, which is too small to contain the tables, indexes and temporary files when creating the database in InnoDB.

Figure 5.7 presents the geometric mean of the average elapsed query times for each system configuration. The average elapsed time for each query (Q1, Q3, Q4, Q6 and 10,000 updates) is calculated by summing the execution times for each query within the thread and dividing either by nine (for the query thread) or by the number of updates completed. The query thread is run to completion; the update thread is stopped once the query thread finishes, and only updates completed while the query thread was still running are counted.

In the figure, the first bar represents the geometric mean for the average elapsed times for the workload on the Intel SSD, and the last bar represents GM of the elapsed times for the workload on the HDD. In between are bars representing the elapsed times for the One Buffer Policy, with the SSD-BM allowed to take up varying amounts of space on the SSD. For example, “Hy16I” means the SSD-BM is 16 GB and is run on the Intel (I) SSD.

If the Intel SSD is used instead of the HDD, performance improves dramatically – the geometric mean of the five query types for the Intel SSD is about 3.5 times less than that of the HDD. However, a) the Intel SSD has much better bandwidth than many SSDs, so this result may not generalize for all SSDs, and b) SSDs are much more expensive than HDDs, and often substantially smaller, so it may be cost-prohibitive to completely replace the HDD(s) with SSDs.

The SSD-BM provides performance between that of the Intel SSD alone and the conventional HDD alone. It is unable to achieve performance equal to or better than that of the Intel SSD alone since the Intel SSD can perform random writes at the same speed as sequential writes, so changing
a random write to a sequential write not only does not help, but actually incurs additional costs since it is then evicting a potentially useful block from the SSD. The performance of the SSD-BM is proportional to its size, which is to be expected, and results in elapsed times that are 15%-50% lower than the geometric mean of the HDD.

All the previous results used the Intel SSD. We repeated the above experiments (without the Hy14 and Hy10 configurations) with the MTRON SSD. As Figure 4.1 shows, the MTRON disk has a lower sustained bandwidth than the Intel disk. The effect of the slower disk can be seen in Figure 5.8: the MTRON SSD takes twice as long to run the workload as the Intel SSD.

Further, the Hy16M SSD-BM configuration has a lower average elapsed time than the MTRON disk alone. This is in contrast to the results with the Intel disk, where the Intel disk consistently outperformed all of the SSD-BM configurations. Additionally, the Hy8M configuration’s elapsed time is almost exactly halfway between that of the SSD-only and the HDD-only. We believe these results show promise if one is only able to afford a less expensive SSD.

5.4.2 Buffer Pool Reuse

An important question to ask when studying caching is the number of times a particular element in the cache is used before being evicted. Figure 5.9 presents the cumulative distribution function for reuse for the Hy16I, Hy14I, Hy12I, Hy10I, Hy8I and Hy4I experiments. For a given (x,y) coordinate on the curves, y% of the blocks were used \( \leq \) x times. As an example, for the Hy16I configuration, 40% of the entries were used zero or one times. In the larger configurations, almost all of the pages are used between one and four times. However, in the Hy4I and Hy8I configurations, only about 15% of pages are reused, and then often only once. Thus, Figure 5.9 clearly shows that the one buffer approach does not do a good job capturing the reuse that occurs in the workload.

These results are for our One Buffer Pool replacement policy, which is a simple circular-queue-based scheme, based on when the page is first written to the SSD. However, this knowledge helps guide our more advanced, Two Buffer Pool replacement policy, since it suggests that pages that have more than a threshold number of uses should be retained, at the cost of lesser-used pages. This
Figure 5.7 Geometric mean of the average elapsed times for each of the queries with different system configurations. The first bar is for the Intel SSD-only system, the middle bars are for different sizes (in GB) of the SSD-BM, and the last bar is for the HDD-only system.

Figure 5.8 Same experiment as the previous figure, but with the MTRON SSD.
preference is implemented by logically partitioning the SSD into two regions. Our study exploring this new policy, its sizing and threshold is presented in the next subsection.

### 5.4.3 Two Buffer Pool Replacement Policy

Section 5.2.3.2 described a different algorithm for managing the SSD-BM: the Two Buffer Pool policy. Figure 5.10 presents a sensitivity analysis for the Two Buffer Pool policy, where the percent of the buffer dedicated to the cold (or new) pages, and the threshold for promoting pages to the hot region are varied. We study five different cold allocations (10%, 25%, 50%, 75% and 80%) and two promotion thresholds (greater than one use, and greater than five) and compare the performance to the One Buffer policy, all using the Intel SSD (MTRON results are presented later). For these experiments, we allocated 8 GB to the SSD-BM, or enough to fit half the working set.

Figure 5.10 shows the geometric mean of the average elapsed times for either the allocation and threshold, or the one buffer case (Hy8I). “10%, >1” means that 10% of the buffer is given to
the cold region (and thus 90% to the hot data), and pages are promoted from the cold region to the hot region if, upon would-be eviction, the page had been used more than one time.

For every allocation, the “$> 5$” threshold has a slightly higher geometric mean for the total result than the “$>1$” threshold. This result makes sense given the results in the previous subsection, which show a very small percentage of pages that are used greater than five times: this threshold is too high. Further, a 50% allocation of the allowed 8 GB leaves only 4 GB, thus pushing the reuse statistics to be similar to those of the Hy4I results, not the Hy8I.

Of the varying sensitivities studied, the “25%, $>1$” configuration gave the best overall performance. This configuration best juggled the desire for short-term reuse (when a page is reused within thousands to the low tens of thousands of SSD-BM accesses), and long-term reuse. Part of the reason that the One Buffer policy is less successful is that we never configured it with enough space to hold all the tables plus the updates, so since Lineitem is accessed by all the queries, the buffer was not large enough to retain pages until their next use. In contrast, the Two Buffer policy only over-writes the first 25% of what was written on the first pass through the SSD-BM. The other 75% of what was first added to the SSD-BM is not overwritten except by hot pages. Thus, the Two Buffer Pool policy is able to capture more long-term reuse. The 10% configurations perform worse since they are not big enough to capture the shorter-term reuse.

We then experimented with the “25%, $>1$” configuration for other SSD-BM sizes. Those results are presented in Figure 5.11. Compared to the One Buffer results, the Two Buffer Pool 12 GB and 8 GB results show significant improvement. However, both the Two Buffer Pool 16 GB and 4 GB results are significantly worse, which suggests that other allocations and thresholds should be tried if the workload either fits entirely into the SSD-BM, or if only a small percentage of the workload fits.

Figure 5.12 presents the results of the same experiments as Figure 5.11, except using the MTRON disk. All of the configurations using the Two Buffer Pool Policy have elapsed times within 10% of the MTRON disk alone. These results are very promising, and are even better than the One Buffer Policy results, since they suggest that if the working set cannot fit on a given
Figure 5.10  Sensitivity analysis for an 8 GB SSD-BM with the Two Buffer Pool replacement policy on the Intel SSD. The percent tells the percent of the SSD-BM allocated to the cold region, and the > tells the threshold number of uses for promotion to the hot region.

Figure 5.11  Geometric mean of average elapsed times for an SSD-BM of varying sizes using the Two Buffer Pool replacement policy on the Intel SSD. 25% of the SSD-BM is allocated to the cold region, and pages are promoted to the hot region if they have been used more than once while in the cold region.
commodity SSD, if the SSD is large enough for only a quarter of the set, one can achieve similar performance to the case when the entire working set can fit in the SSD.

**5.4.4 Multiple Update Streams**

The next group of experiments is for a workload similar to the one just discussed, but with two update streams. Comparatively, it gets worse overall performance than the previous workload due to increased interference between threads and because the second update thread update records across the entire table, rather than in a focused area. The Two Buffer Pool and One Buffer Pool results are presented side-by-side in Figure 5.13 using the MTRON disk.

Notice that the additional update stream has a far greater impact for the HDD-only system than the MTRON-only system: the MTRON system’s GM increases by roughly 200 seconds (a 50% increase), whereas the HDD system’s GM increased by over 500 seconds (a 70% increase). The 16GB One Buffer SSD-BM policy again gives comparable performance to the SSD-only system, however, there is now more variance amongst the different configurations. The worst performing SSD-BM system still outperforms the HDD-only system by about 20%. It is also interesting to note that the Two Buffer Pool policy either performs comparably to or worse than the One Buffer policy and the 8 GB SSD-BM configurations perform worse than the smaller 4 GB configurations.

**5.4.5 Skewed Data**

Figure 5.14 shows the same information as Figure 5.7, and also use the Intel SSD, but the results are for the skewed data set and update stream. Compared to Figure 5.7, there is much less variance among the different SSD-BM configurations, which suggests there was substantially more short-term reuse (else the Hy4I configuration would have performed much worse than the Hy16I configuration). The fact that the HDD-only system benefited more than the SSD-only system suggests that the queries had less disk-head movement than the queries in the other workloads.

The flattening is even more noticeable in Figure 5.15, which shows the elapsed times for the Two Buffer Pool policy on the skewed data, with varying allocations and thresholds, on an 8 GB SSD-BM. The 25% cold region benefits from a lower threshold for promotion, suggesting that there
Figure 5.12  Geometric mean of average elapsed times for an SSD-BM of varying sizes using the Two Buffer Pool replacement policy on the MTRON SSD. 25% of the SSD-BM is allocated to the cold region, and pages are promoted to the hot region if they have been used more than once while in the cold region.

Figure 5.13  Geometric mean of average elapsed times for the SSD-BM running a workload with one query thread and two update threads on the MTRON SSD.
are a number of pages that are used between one and five times in the span of 62,500 (8 GB/32 KB pages = 250,000 * 25%) accesses. By virtue of being larger, the 50% and 75% configurations would be able to capture this behavior, since it takes more accesses to revisit a specific SSD-BM frame when the cold region is larger. However, even the best Two Buffer configuration, “75%, > 1” has performance comparable to the One Buffer case.

5.5 Related Work

There is a variety of literature exploring how to take advantage of the performance characteristics of SSDs. We discuss the related work in three main areas: combining flash and HDD; replacing HDDs with SSDs; and the buffer management literature.

5.5.1 Flash and Hard Disk Drives, Together

Samsung’s hybrid drive combines a standard disk with less than 1 GB of flash [5]. The flash memory caches data to and from the HDD, to decrease power consumption, to increase reliability through fewer seeks, and to allow faster start up times.

Koltsidas and Viglas explore using both a SSD and HDD in one system in order to take advantage of each drive’s relative strengths [35]. They develop a storage system and page buffer replacement strategy to place pages on the most appropriate medium: frequently updated pages go to HDD, while read-heavy pages go to SSD. Since pages evicted from the buffer pool could go to either the HDD or the SSD, it was necessary to create a page replacement algorithm that evicts pages accordingly. Additionally, a page’s workload can vary from read-intensive to write-intensive over time, so the authors created and tested three page migration algorithms, a conservative, aggressive and hybrid scheme, to ensure each page was located on the best disk. Their results show that their algorithms do a good job of placing pages on the appropriate medium, thus resulting in performance similar to the minimum of the HDD or SSD latency.

In “The five-minute rule, twenty years later,” Graefe argues that databases will use flash memory as an extended persistent store, in addition to disks [24]. Graefe argues why he believes this is the case, but he does not provide an implementation and performance results.
Figure 5.14  Geometric mean of average elapsed times for the queries over the skewed tables. The SSD-BM uses the One Buffer Pool replacement policy and the Intel SSD.

Figure 5.15  Geometric mean of average elapsed times for the queries over the skewed tables. The SSD-BM uses the Two Buffer Pool replacement policy and the Intel SSD; the 8 GB SSD-BM configuration is also included for reference.
5.5.2 Disk Drive Replacement

Myers [44] and Lee [39] examine the performance of databases stored on SSDs rather than HDDs drives in an unmodified database. [39] finds that this substitution can greatly improve throughput and response time for the transaction log and rollback segments; sort-merge and hash join queries are helped by replacing a magnetic temp drive with a solid-state temp drive. [44] suggests that SSDs did not provide high enough performance for non-random reads to justify their widespread use, but they provide significant performance gains when using secondary indexes and index-based joins.

Several recent papers present new algorithms designed to take advantage of flash drives’ strengths. Wu [63] creates a software layer on top of a SSD’s flash translation layer (FTL) in order to improve B-tree performance and reduce wear on the SSD. Lee [38] avoids small random write requests by buffering a page’s modifications in memory on a per-page basis. These change logs are then written to a log area in flash memory and eventually merged to the database. Ross [55] presents new algorithms for counting, linked lists and B-trees that minimize new writes by taking into account the fact flash blocks are cleared as one and are write-once. Researchers at HP Labs have created new query processing techniques, specifically scans and joins, that use a SSD’s superior random read performance on small pages to decrease response times [56, 60]. Nath introduces B-FILE, which efficiently stores large samples on flash [46]. Finally, Li [42] proposes the FD-tree, a variant of the B-tree that hierarchically stores the data, and only performs random writes to the top-level tree.

In the systems community, Narayanan et al study the cost-performance trade-off for SSDs both completely replacing HDDs and acting as an intermediate caching tier [45]. Their intermediate caching tier is either a simple write-ahead log or a combined write-ahead log/ read cache. The combined read/ write tier uses either an LRU or long-term random access frequency replacement policy. All results are generated using storage-system level traces and a simple I/O cost model, and results are presented in terms of performance per dollar. They found that, in general, SSDs do not provide enough capacity per dollar to give a cost-effective improvement in system performance.
5.5.3 Buffer Pool Management

Managing the buffer pool has been extensively studied within the database community. Chou and DeWitt introduced DBMIN, which used the knowledge of specific common database access patterns, such as sequential reads in a scan, to help allocate pages in the buffer pool [17]. Later, O’Neil et al introduce LRU/k, an improvement on traditional LRU (or LRU/1, in this model) [48]. It gives priority to buffer pages based on the k-th most recent access. Johnson [32] uses LRU/k to develop the “Two Queue” algorithm, which performs as well as LRU/2 but does not require tuning. A more recent idea combines LRU and LFU policies to get the LRFU policy [37].

There have been two recent papers outside the database community that study different buffer replacement policies with flash. Within the embedded systems community, Park et al presented a modified LRU algorithm for use in file system caches [49]. Park avoids random writes to the backing storage (flash memory) by logically splitting the LRU list into two sections: the working region, and the clean-first region. The policy preferentially evicts pages from the clean-first region in order to avoid random writes. The algorithms reduce replacement cost in the swap system by 28% and in the buffer cache by 26%. This research is orthogonal to our own, since its focus is adapting the replacement algorithm in the file system’s primary storage (memory). Kim also presents a modified LRU policy for file systems that reduces the costs of random writes to SSDs [33]. In BPLRU, a RAM write buffer is added inside the SSD and works at the block level.

5.6 Future Work

There are a number of avenues for future research. One avenue is using different SSDs, as there are higher-performance SSDs available. Thus, we plan to study SSD-BM with a FusionIO board [4]. Another important area of future research is studying the replacement policy. We have only presented two options, but there others, including changing the primary buffer pool’s eviction policy (as in [35, 49]). Additionally, we have only studied a relatively simple workload; we believe other workloads can benefit from the addition of the SSD-BM. And, finally, we would
like to perform a performance/cost comparison of the SSD-BM compared to the SSD alone and HDD alone systems, similar to that in [45].

5.7 Conclusions

In this chapter, we present a novel way to extend the buffer pool using a SSD. If the commodity SSD were treated just like a disk or RAM, we would have found few, if any, gains in performance. However, we presented an algorithm to treat the SSD like a circular queue, thus avoiding the performance penalties of random writes to the SSD. In our modified TPC-H workload, the geometric means of the queries were reduced to half to three-quarters the baseline. Further, with the less advanced of the SSDs, our scheme was able to beat or come to within 10% the performance of the SSD alone for one workload, even when our scheme used only a quarter the space on the SSD. These results are particularly promising since SSDs are so expensive compared to HDDs: if the full workload cannot fit on the SSD, one might be able to use SSD-BM to achieve performance very close to that of the SSD alone.
Chapter 6

Conclusion

In this dissertation, I have presented techniques to take advantage of current hardware and explored different ways to physically store data, depending on the system. I will now discuss the main contributions of this dissertation, current and future applications and directions, and some final words.

6.1 Contributions

The contributions of this dissertation fall into two main categories: studying compression in databases, and novel uses for solid-state disks to improve database performance.

In Chapter 2, we experimentally verified some assumptions about the use of compression in a DBMS and found that, thanks to fast processors, some of them no longer hold true. Further, we introduced two new techniques to speed up the decoding of Huffman-encoded attributes. Finally, we quantified the decompression costs for different compression formats, and outlined a way to analyze the system and compression trade-offs in order to achieve the best performance.

Chapter 3 continued our study of compression and expanded it to also examine row store performance compared to column store performance. We studied a wider range of queries than previous studies had and used more aggressive compression. Two interesting results are that, for the types of compression we studied, row stores and column stores compress equally well. This is an important result, since column store proponents argue columns compress better than row stores. Previous studies had also only examined very simple queries, and our results show that adding
more processor-intensive tasks, such as additional predicates and more qualifying tuples, can tip the balance back in favor of row stores.

In Chapters 2 and 3, we adapt databases to use compression in order to help re-balance processors and disks. In the next two chapters, we focused on another hardware trend: the introduction of solid-state disks.

Chapter 4 introduced Dual-Drive, a system that combines a solid-state disk and a hard disk into one logical unit to help overcome our solid-state disk’s poor write characteristics, while still taking advantage of its fast random reads. This device has its flaws and drawbacks, but, since solid-state disks are still rapidly evolving, we believe it is best to explore a wide range of SSD adaptations, particularly since DD could provide a 30% performance improvement for some workloads, at a roughly 10% increase in price.

Finally, in Chapter 5, we developed another technique for using solid-state disks: the solid-state disk buffer manager (SSD-BM). For the faster Intel SSD, the SSD-BM did not provide much of a performance gain, but, if one only had enough money to store part of their working set on a fast SSD, the SSD-BM should be considered. And, if one had a lower-end, commodity SSD, SSD-BM can provide performance better than the SSD alone for some configurations, or within 10% of the SSD-alone, even when only one quarter of the workload fit on the SSD.

6.2 Applications and Future Directions

We suggest areas for future research in every chapter of this dissertation, but many of the suggestions do not look far into the future. However, I believe many of the ideas can and should be broadened, applied, and studied.

My first two chapters focused on the trade-offs of compression and different storage paradigms. No single compression scheme or storage paradigm dominated the others, even for a given system or workload. However, given the possible performance gains to be had if the best scheme were always chosen, an interesting area for future work would be to automate the decision. Further, we only studied a small number of compression algorithms, systems and workloads; a model to help guide the choice would be an invaluable tool for database administrators.
The last two chapters focused on novel uses for solid-state disks. SSDs are a promising avenue for future research, but their performance characteristics vary enormously from one device to the next. I have presented some options for current disks, but I think future disks will behave differently and might have a different interface altogether. The performance characteristics of the higher-end device we studied, the Intel SSD, bode well for SSDs in general, and might also lead to interesting new database designs, particularly due to its symmetries for both sequential and random reads or writes.

Though I discuss the processor trends – that their performance is improving at a faster rate than those of memory and disks, and the move toward multi-core – I have not studied other ways to redesign the database engine and storage systems for them. Future processors will require more multithreading to achieve the best performance. While some of the threads can be used for compression and decompression, we will need more techniques like those of processing pages independently (discussed in Chapter 2). Multi-core processors are here to stay, and we, as software developers, must adapt.

6.3 Closing Remarks

Technology is always changing, so this area – of adapting databases for new hardware – will continue to be an important area of research and future performance improvements. My work provides some possibilities for now, but it is likely that in five years (or even two or three), these approaches will not yield the same conclusions. I look forward to seeing how technology continues to evolve, and figuring out ways to take advantage of it.
LIST OF REFERENCES


