

COMP 421: Files & Databases

Lecture 4: It's log!

Announcements

Project 1: Buffer Pool Manager
will be released this afternoon

Important Dates:

- Buffer pool manager lecture [9/10]
- Bootcamp 2 [9/16, 9/17]
- Project 1 Due [9/29], late days allowed
- Read the description, start early, come to OH

P0 Postmortem

- Median was 100, Mean was 91.7
- Some FAQs
 - devcontainer?
 - Where does my code go?
 - Build/debug/test?
- Mostly, we wanted you to figure this out, going forward, we'll be better about file paths
- Formatting issues
 - See writeup, `make` targets for formatting
 - Some Windows users had issues, check VS Code settings
- If you start early and come to OH, we can help!

Bootcamp 1 Postmortem

- Bootcamp 2 will be longer so we don't rush
 - 1.5 hours, rooms TBD
- More interactive, more demos, more writing code
- Not an explicit walkthrough of P1, but highly related
 - If you have read and started P1, you will get more out of Bootcamp
 - We can answer specific C++ questions

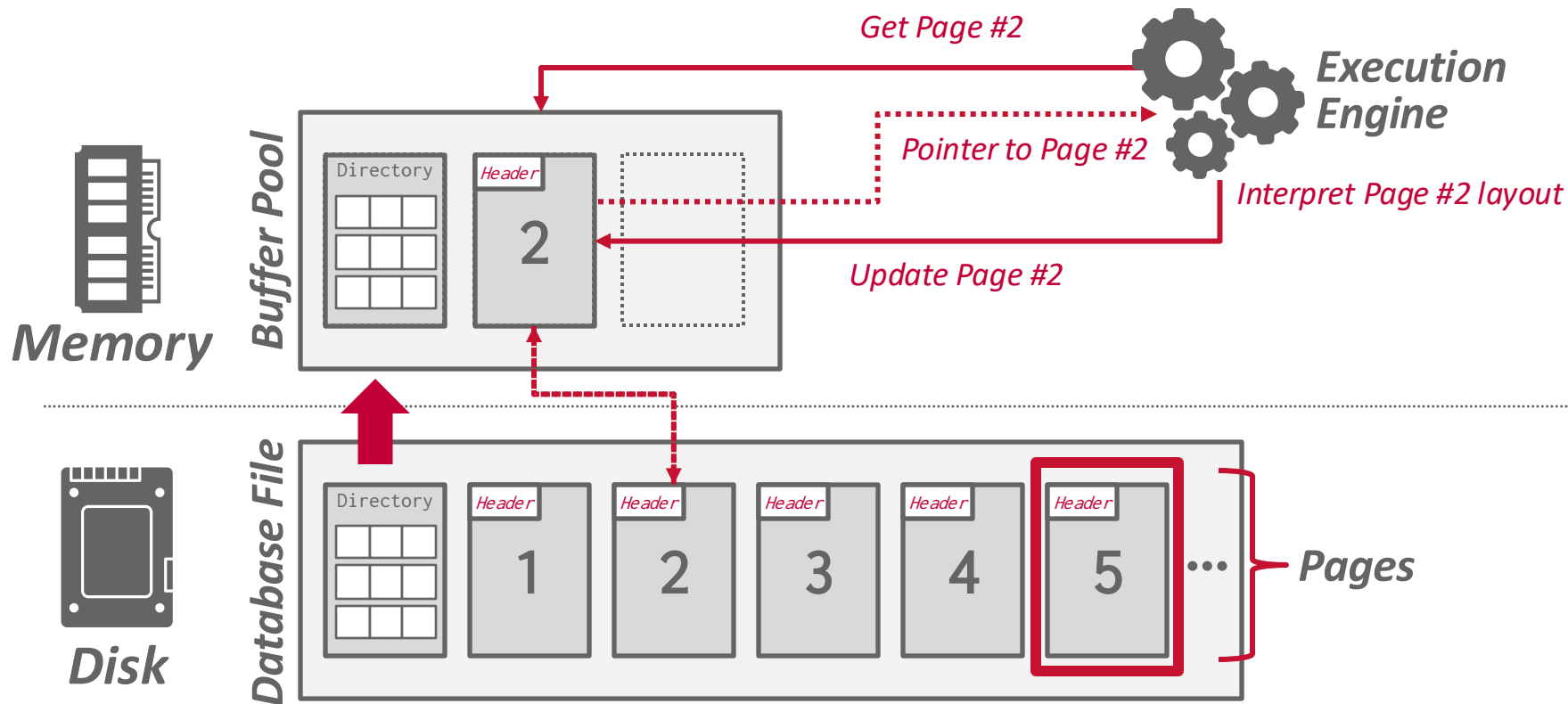
Last Class

We presented a disk-oriented architecture where the DBMS assumes that the primary storage location of the database is on non-volatile disk.

We then discussed a page-oriented storage scheme for organizing tuples across heap files.

We had just gotten to talking about how to organize bytes within a page...

Disk-oriented DBMS



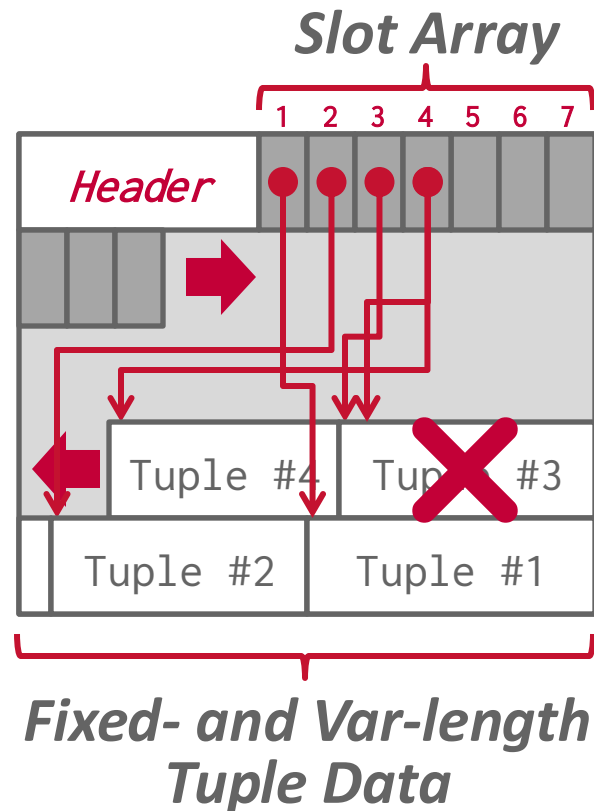
Slotted Pages

The most common layout scheme is called slotted pages.

The slot array maps "slots" to the tuples' starting position offsets.

The header keeps track of:

- The # of used slots
- The offset of the starting location of the last slot used.



Record IDs

The DBMS assigns each logical tuple a unique record identifier that represents its physical location in the database.

- File Id, Page Id, Slot #
- Most DBMSs do not store ids in tuple.
- SQLite uses ROWID as the true primary key and stores them as a hidden attribute.

Applications should never rely on these IDs to mean anything.

 PostgreSQL
CTID (6-bytes)

 SQLite
ROWID (8-bytes)

 Microsoft SQL Server
%%physloc%% (8-bytes)

ORACLE®
ROWID (10-bytes)

Tuple-oriented Storage

Insert a new tuple:

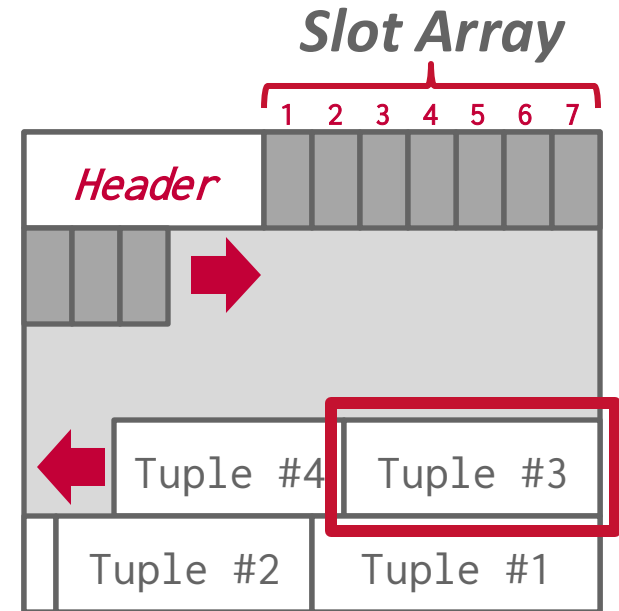
- Check page directory to find a page with a free slot.
- Retrieve the page from disk (if not in memory).
- Check slot array to find empty space in page that will fit.

Update an existing tuple using its record id:

- Check page directory to find location of page.
- Retrieve the page from disk (if not in memory).
- Find offset in page using slot array.
- If new data fits, overwrite existing data.
Otherwise, mark existing tuple as deleted and insert new version in a different page.

Today's Agenda

Tuple Structure
Log-Structured Storage




Tuple Storage

A tuple is essentially a sequence of bytes prefixed with a header that contains meta-data about it.

It is the job of the DBMS to interpret those bytes into attribute types and values.

The DBMS's catalogs contain the schema information about tables that the system uses to figure out the tuple's layout.

Data Layout



```
CREATE TABLE foo (  
  id INT PRIMARY KEY,  
  value BIGINT  
);
```



unsigned char[]

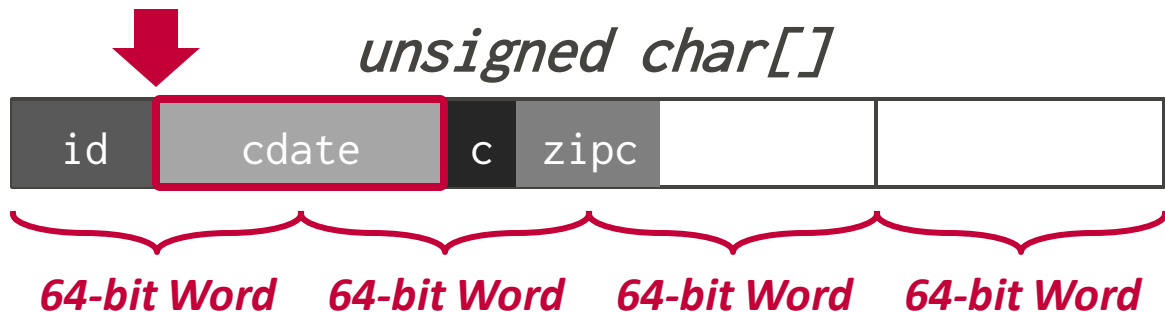


```
reinterpret_cast<int32_t*>(address)
```

Word-aligned Tuples

All attributes in a tuple must be word aligned to enable the CPU to access it without any unexpected behavior or additional work.

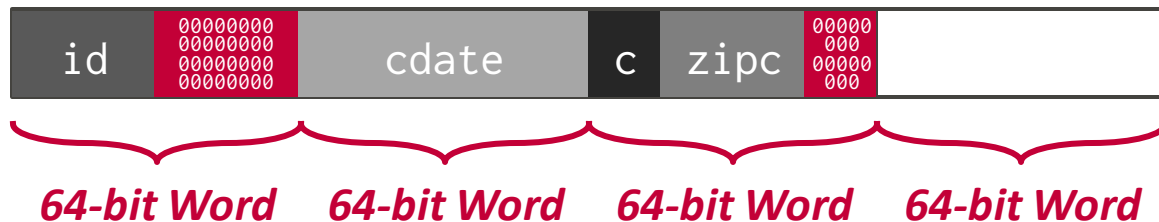
```
CREATE TABLE foo (  
32-bits id INT PRIMARY KEY,  
64-bits cdate TIMESTAMP,  
16-bits color CHAR(2),  
32-bits zipcode INT  
);
```



Word-alignment: Padding

Add empty bits after attributes to ensure that tuple is word aligned. Essentially round up the storage size of types to the next largest word size.

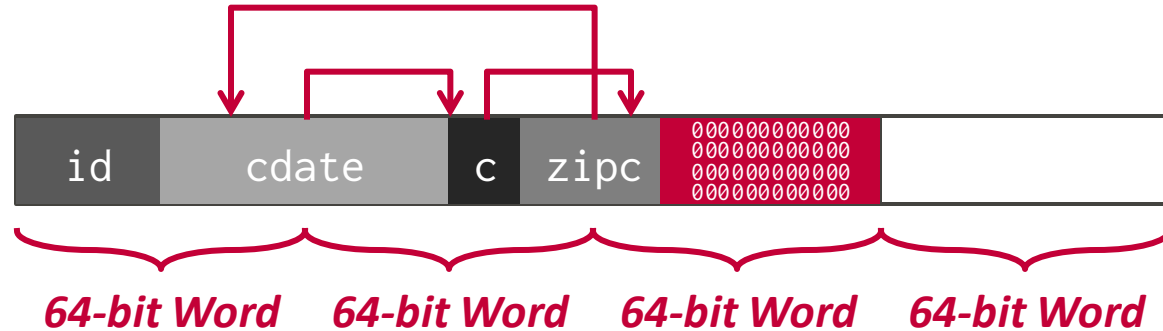
```
CREATE TABLE foo (  
  32-bits id INT PRIMARY KEY,  
  64-bits cdate TIMESTAMP,  
  16-bits color CHAR(2),  
  32-bits zipcode INT  
);
```



Word-alignment: Reordering

Switch the order of attributes in the tuples' physical layout to make sure they are aligned.
→ May still have to use padding to fill remaining space.

```
CREATE TABLE foo (  
32-bits id INT PRIMARY KEY,  
64-bits cdate TIMESTAMP,  
16-bits color CHAR(2),  
32-bits zipcode INT  
);
```



Data Representation

INTEGER/BIGINT/SMALLINT/TINYINT

→ Same as in C/C++.

FLOAT/REAL vs. NUMERIC/DECIMAL

→ IEEE-754 Standard / Fixed-point Decimals.

VARCHAR/VARBINARY/TEXT/BLOB

→ Header with length, followed by data bytes **OR** pointer to another page/offset with data.

→ Need to worry about collations / sorting.

TIME/DATE/TIMESTAMP/INTERVAL

→ 32/64-bit integer of (micro/milli)-seconds since Unix epoch (January 1st, 1970).

Variable Precision Numbers

Inexact, variable-precision numeric type that uses the "native" C/C++ types.

Store directly as specified by IEEE-754.

→ Example: **FLOAT**, **REAL**/**DOUBLE**

These types are typically faster than fixed precision numbers because CPU ISA's (Xeon, Arm) have instructions / registers to support them.

But they do not guarantee exact values...

Variable Precision Numbers

Rounding Example

```
#include <stdio.h>

int main() {
    #include <stdio.h>

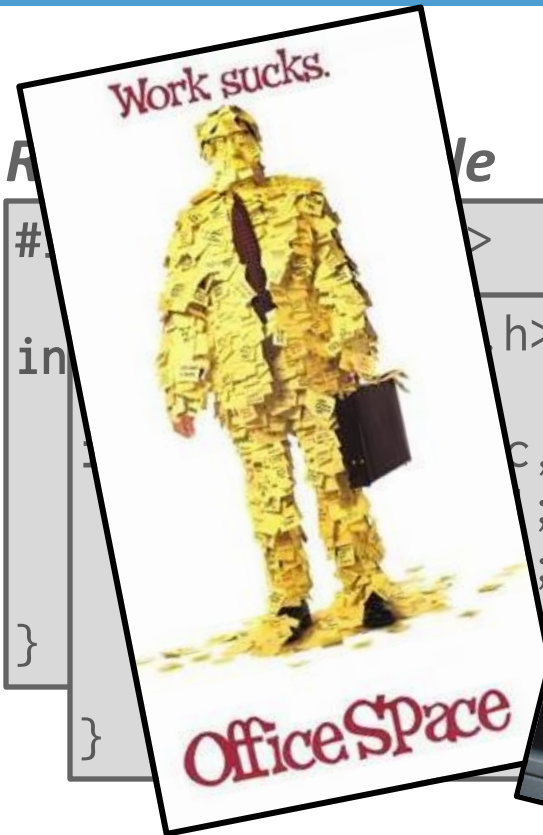
    int main(int argc, char* argv[]) {
        float x = 0.1;
        float y = 0.2;
        printf("x+y = %.20f\n", x+y);
        printf("0.3 = %.20f\n", 0.3);
    }
}
```

Output

```
x+y = 0.300000
0.3 = 0.300000
```

```
x+y = 0.30000001192092895508
0.3 = 0.29999999999999998890
```

Variable Precision Numbers



Output

$x+y = 0.300000$

$0.3 = 0.300000$

$x+y = 0.30000001192092895508$

$0.3 = 0.29999999999999998890$



Fixed Precision Numbers

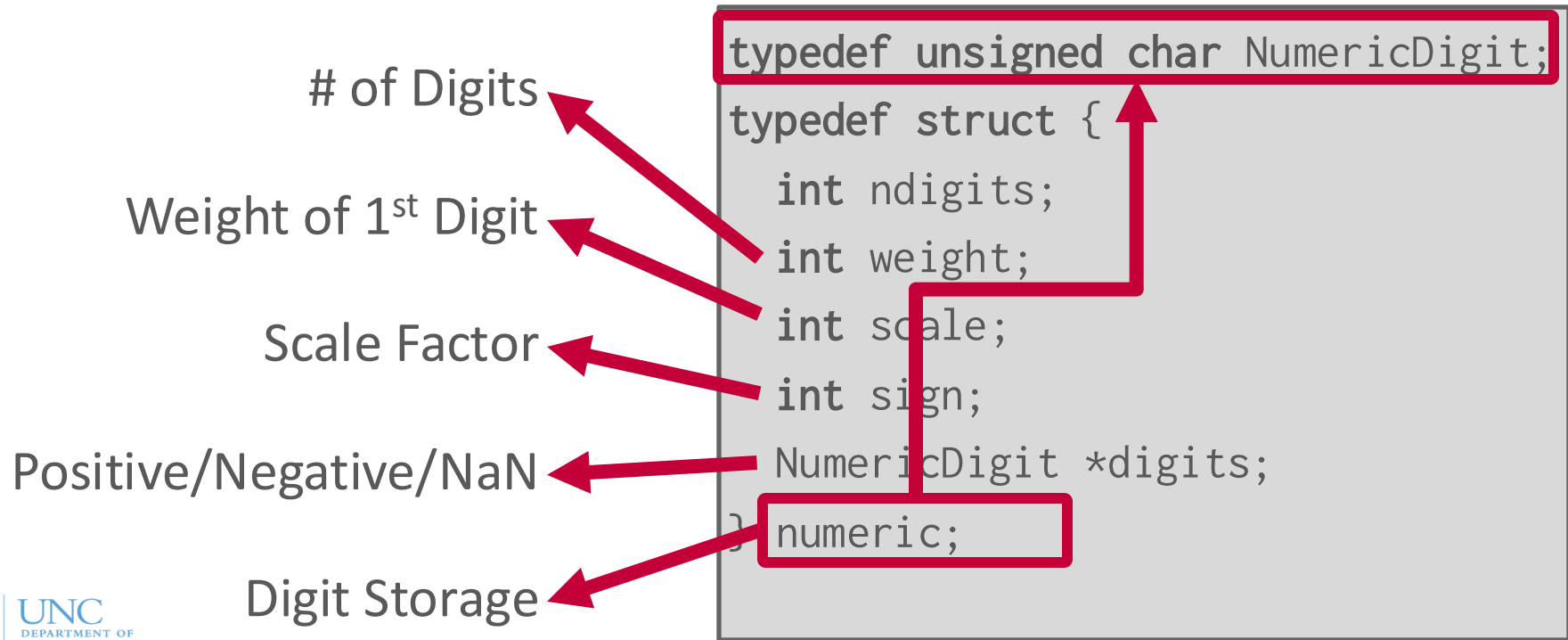
Numeric data types with (potentially) arbitrary precision and scale. Used when rounding errors are unacceptable.

→ Example: **NUMERIC**, **DECIMAL**

Many different implementations.

- Example: Store in an exact, variable-length binary representation with additional meta-data.
- Can be less expensive if the DBMS does not provide arbitrary precision (e.g., decimal point can be in a different position per value).

Postgres: NUMERIC



NULL Data Type

Choice #1: Null Column Bitmap

- Store a bitmap in a centralized header; all null attributes are null.
- This is the most common approach

Choice #2: Special Values

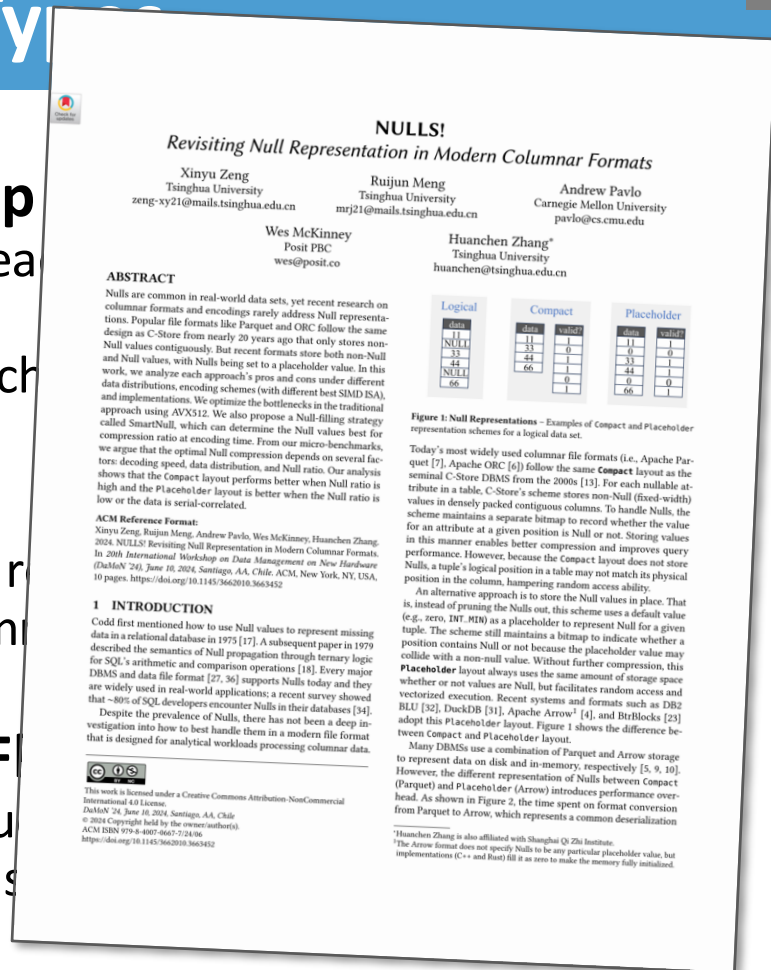
- Designate a placeholder value to represent null (e.g., INT32_MIN). More common in older DBMS.



**Don't
Do This!**

Choice #3: Per Attribute Null Flag

- Store a flag that marks that a value is null.
- Must use more space than just a special value, as it messes up with word alignment.



Large Values

Most DBMSs do not allow a tuple to exceed the size of a single page.

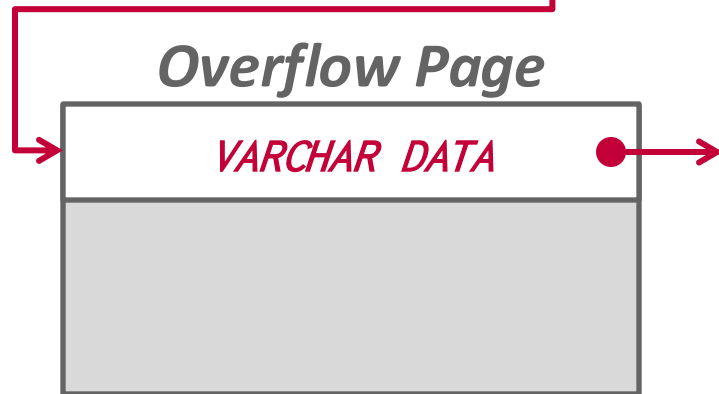
To store values that are larger than a page, the DBMS uses separate **overflow** storage pages.

- Postgres: TOAST (>2KB)
- MySQL: Overflow (>½ size of page)
- SQL Server: Overflow (>size of page)

Lots of potential optimizations:

- Overflow Compression, German Strings

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  data INT,
  contents TEXT
);
```



External Value Storage

Some systems allow you to store a large value in an external file.

Treated as a **BLOB** type.

→ Oracle: **BFILE** data type

→ Microsoft: **FILESTREAM** data type

The DBMS cannot manipulate the contents of an external file.

→ No durability protections.

→ No transaction protections.

To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem?

Russell Sears¹, Catharine van Ingen¹, Jim Gray¹
1: Microsoft Research, 2: University of California at Berkeley
sears@cs.berkeley.edu, vanIngen@microsoft.com, gray@microsoft.com
MSR-TR-2006-45
April 2006 Revised June 2006

Abstract

Application designers must decide whether to store large objects (BLOBs) in a filesystem or in a database. Generally, this decision is based on factors such as application simplicity or manageability. Often, system performance affects these factors.

Folklore tells us that databases efficiently handle large numbers of small objects, while filesystems are more efficient for large objects. Where is the break-even point? When is accessing a BLOB stored as a file cheaper than accessing a BLOB stored as a database record?

Of course, this depends on the particular filesystem, database system, and workload in question. This study shows that when comparing the NTFS file system and SQL Server 2005 database system on a `create, (read, replace), delete` workload, BLOBs smaller than 256KB are more efficiently handled by SQL Server, while NTFS is more efficient BLOBs larger than 1MB. Of course, this break-even point will vary among different database systems, filesystems, and workloads.

By measuring the performance of a storage server workload typical of web applications which use `get/put` protocols such as WebDAV [WebDAV], we found that the break-even point depends on many factors. However, our experiments suggest that *storage age*, the ratio of bytes in deleted or replaced objects to bytes in live objects, is dominant. As storage age increases, fragmentation tends to increase. The filesystem we study has better fragmentation control than the database we used, suggesting the database system would benefit from incorporating ideas from filesystem architecture. Conversely, filesystem performance may be improved by using database techniques to handle small files.

Surprisingly, for these studies, when average object size is held constant, the distribution of object sizes did not significantly affect performance. We also found that, in addition to low percentage free space, a low ratio of free space to average object size leads to fragmentation and performance degradation.

1. Introduction

Application data objects are getting larger as digital media becomes ubiquitous. Furthermore, the increasing popularity of web services and other network applications means that systems that once managed static archives of “finished” objects now manage frequently modified versions of application data as it is being created and updated. Rather than updating these objects, the archive either stores multiple versions of the objects (the V of WebDAV stands for “versioning”), or simply does wholesale replacement (as in SharePoint Team Services [SharePoint]).

Application designers have the choice of storing large objects as files in the filesystem, as BLOBs (binary large objects) in a database, or as a combination of both. Only folklore is available regarding the tradeoffs—often the design decision is based on which technology the designer knows best. Most designers will tell you that a database is probably best for small binary objects and that that files are best for large objects. But, what is the break-even point? What are the tradeoffs?

This article characterizes the performance of an abstracted write-intensive web application that deals with relatively large objects. Two versions of the system are compared; one uses a relational database to store large objects, while the other version stores the objects as files in the filesystem. We measure how performance changes over time as the storage becomes fragmented. The article concludes by describing and quantifying the factors that a designer should consider when picking a storage system. It also suggests filesystem and database improvements for large object support.

One surprising (to us at least) conclusion of our work is that storage fragmentation is the main determinant of the break-even point in the tradeoff. Therefore, much of our work and much of this article focuses on storage fragmentation issues. In essence, filesystems seem to have better fragmentation handling than databases and this drives the break-even point down from about 1MB to about 256KB.

The Trouble with Tuples



Tuple-oriented Storage

Problem #1: Fragmentation

→ Pages are not fully utilized (unusable space, empty slots).

Problem #2: Useless Disk I/O

→ DBMS must fetch entire page to update one tuple.

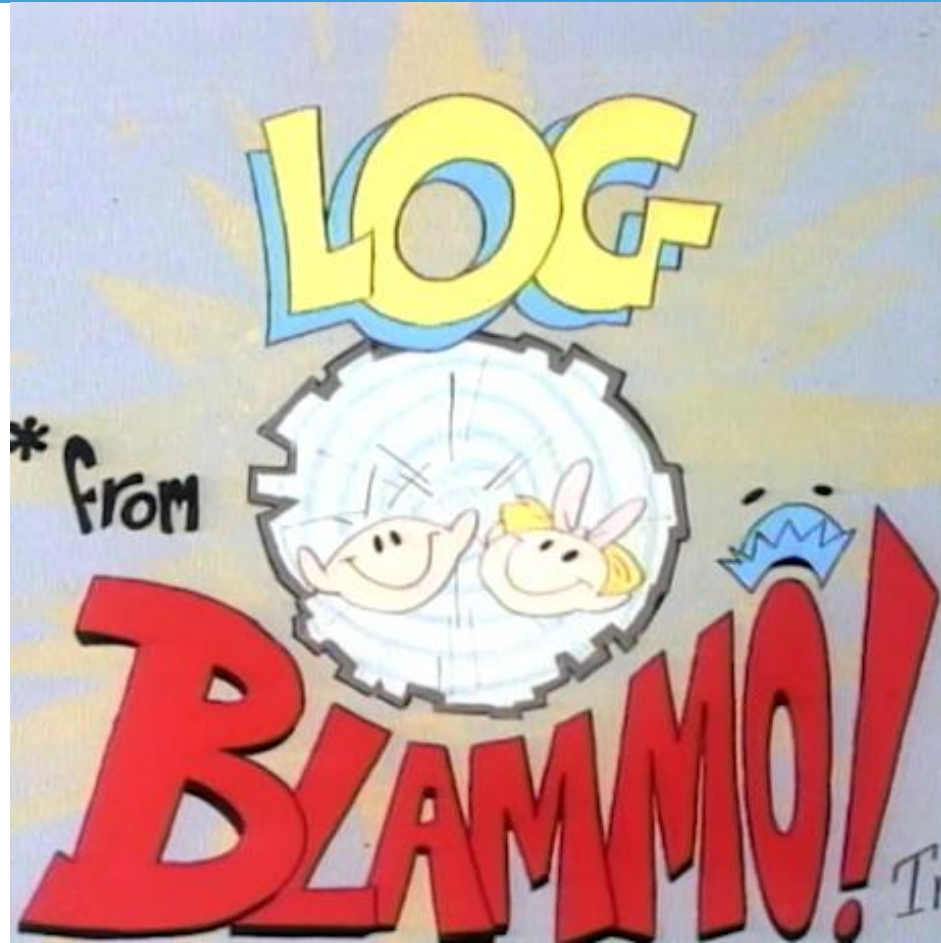
Problem #3: Random Disk I/O

→ Worse case scenario when updating multiple tuples is that each tuple is on a separate page.

What if the DBMS cannot overwrite data in pages and could only create new pages?

→ Examples: Some object stores, [HDFS](#), [Google Colossus](#)

It's Log!



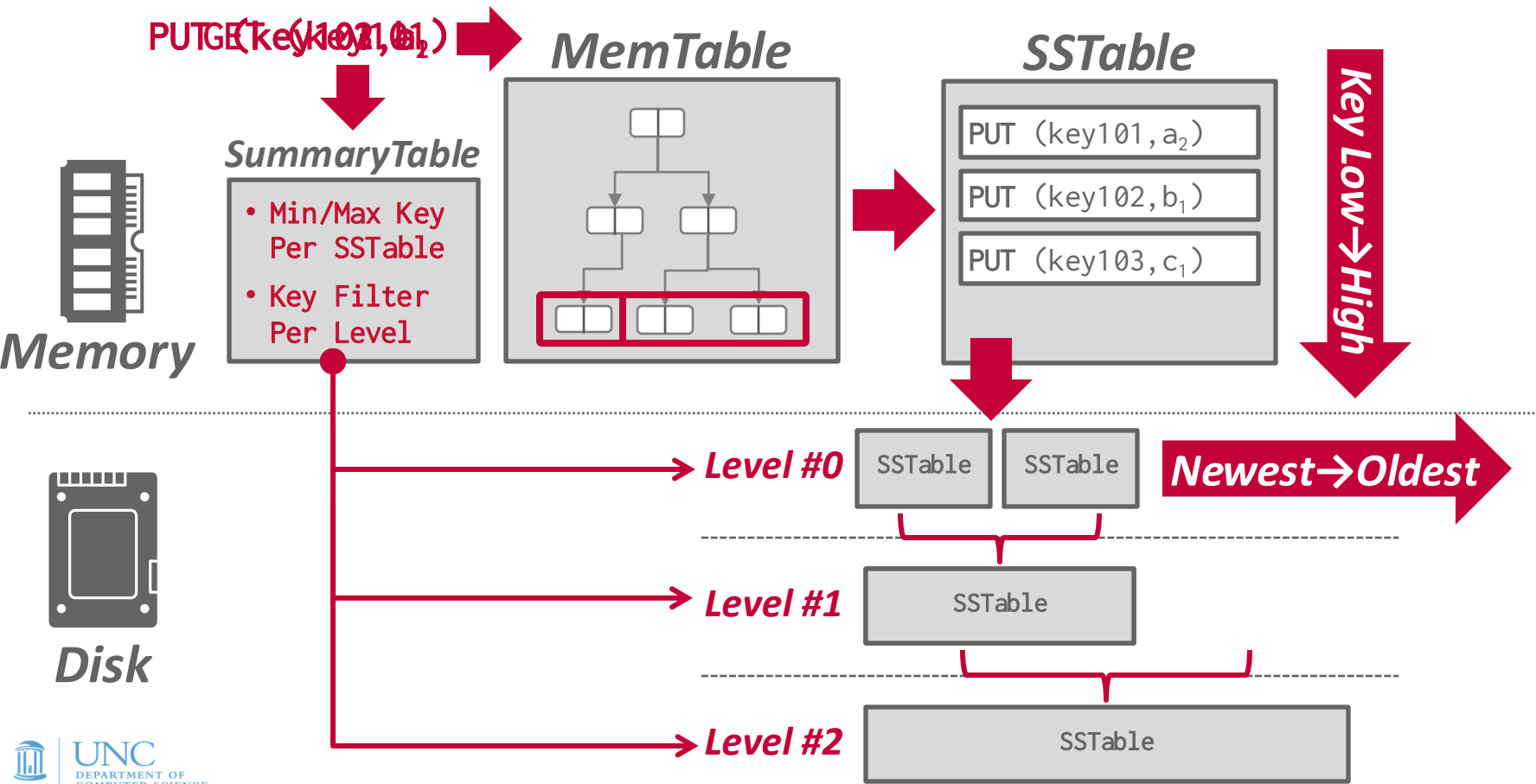
Log-structured Storage

Instead of storing tuples in pages and updating them in-place, the DBMS maintains a log that records changes to tuples.

- Each log entry represents a tuple **PUT/DELETE** operation.
- Originally proposed as log-structure merge trees (LSM Trees) in 1996.

The DBMS applies changes to an in-memory data structure (***MemTable***) and then writes out the changes sequentially to disk (***SSTable***).

Log-structured Storage

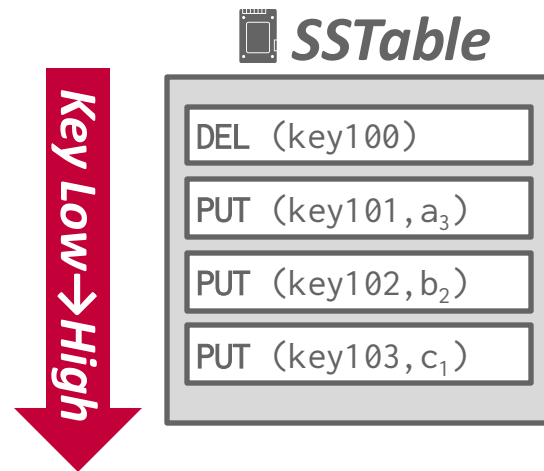


Log-structured Storage

Key-value storage that appends log records on disk to represent changes to tuples (**PUT**, **DELETE**).

- Each log record must contain the tuple's unique identifier.
- Put records contain the tuple contents.
- Deletes marks the tuple as deleted.

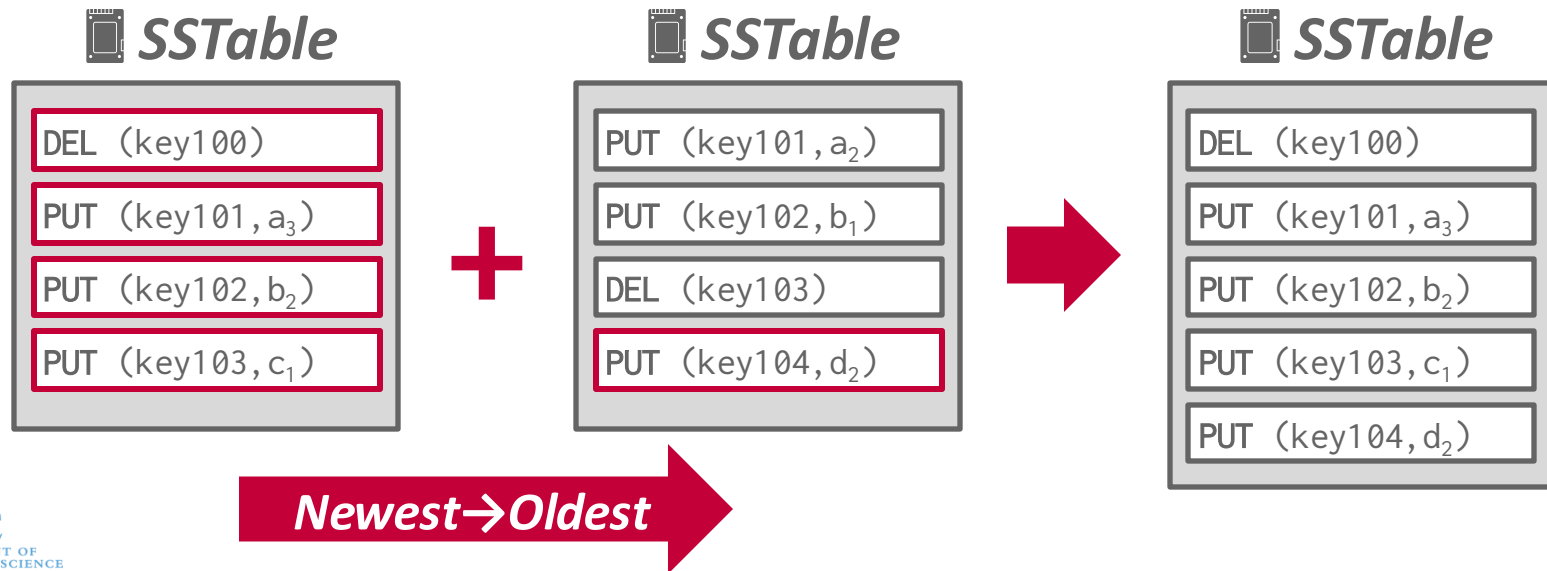
As the application makes changes to the database, the DBMS appends log records to the end of the file without checking previous log records.



Log-structured Compaction

Periodically compact SSTables to reduce wasted space and speed up reads.

→ Only keep the "latest" values for each key using a sort-merge algorithm.



Discussion

Log-structured storage managers are more common today than in previous decades.

→ This is partly due to the proliferation of RocksDB.



RocksDB



levelDB

APACHE
HBASE



yugabyteDB



fauna



TiDB



ClickHouse



CockroachDB



cassandra

WIREDTIGER



NEON

What are some downsides of this approach?

→ Read Amplification

→ Write Amplification

→ Compaction is Expensive

Conclusion

Log-structured storage is an alternative approach to the tuple-oriented architecture.

→ Ideal for write-heavy workloads because it maximizes sequential disk I/O.

Next Class

How to make pages stored on disk available in memory? The buffer pool manager!