Carnegie Mellon University

Intro to Database Systems (15-445/645)

Lecture #04

Database Storage *Part 2*

SPRING 2024 ❯❯ Prof. Jignesh Patel

ADMINISTRIVIA

Homework #1 is due February 2nd @ 11:59pm.

Project #1 is due February 18th @ 11:59pm.

Please **sign the Course Collaboration Policy** on Gradescope if you haven't done so yet.

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.

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 the existing tuple as deleted and insert a new version on a different page.

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 can only create new pages?

→ Examples: Some object stores, HDFS

TODAY'S AGENDA

Log-Structured Storage

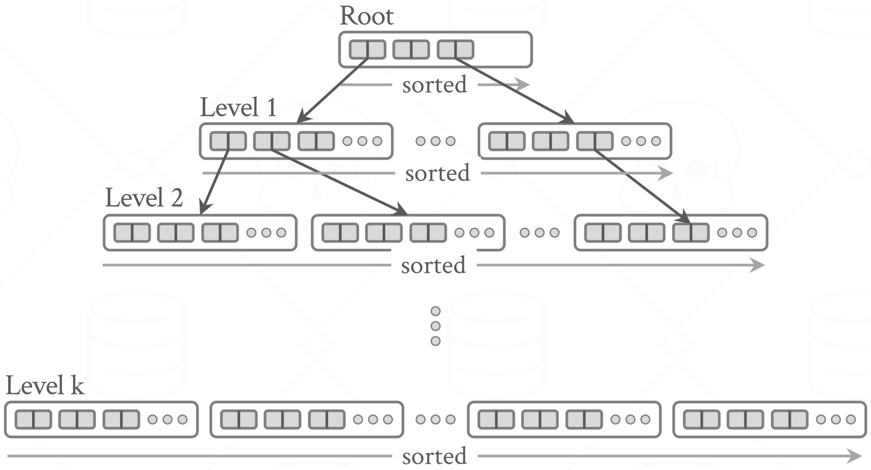
Index-Organized Storage

Data Representation

TREES IN DATABASE STORAGE

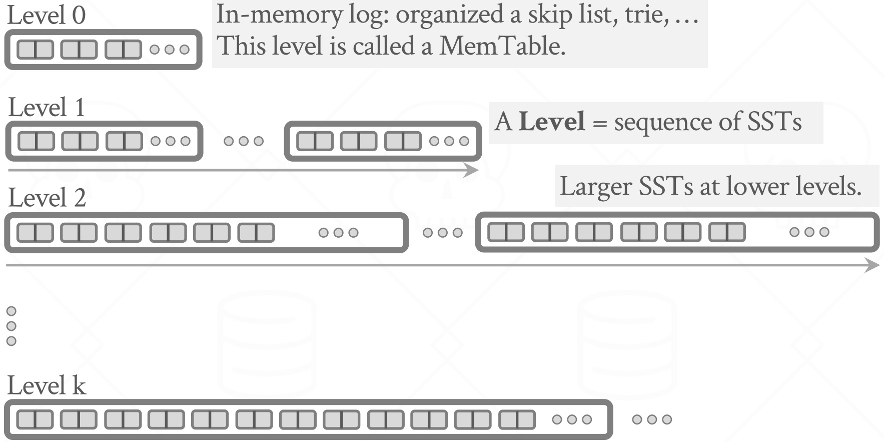
B-tree: Ubiquitous in database systems

- → Balanced tree. Node = page.
- → Same page size (**KBs**) in size across the tree.
- → O(log(n)) for search, insert, delete.
- → Entries: key-value (KV) pairs.
- → Values could be record id, or tuple.
- → Pointers across nodes across the levels.
- → Writes may update multiple pages.



Log-structured Storage: Based on Log-Structured File System (LSFS) by Rosenblum & Ousterhout'92 and Log-structured Merge Trees (LSM Tree) by O'Neil, Cheng, & Gawlick'96

- → Write to a sequentially-growing log rather an update in-place.
- → Flush logs to SSTs (many **MBs** in size). Merge logs periodically.
- → Entries: key-value pairs. Values are records.
- → No pointers across SSTs. SST size grows as the levels increase.
- → Writes are fast, but reads may be slow.

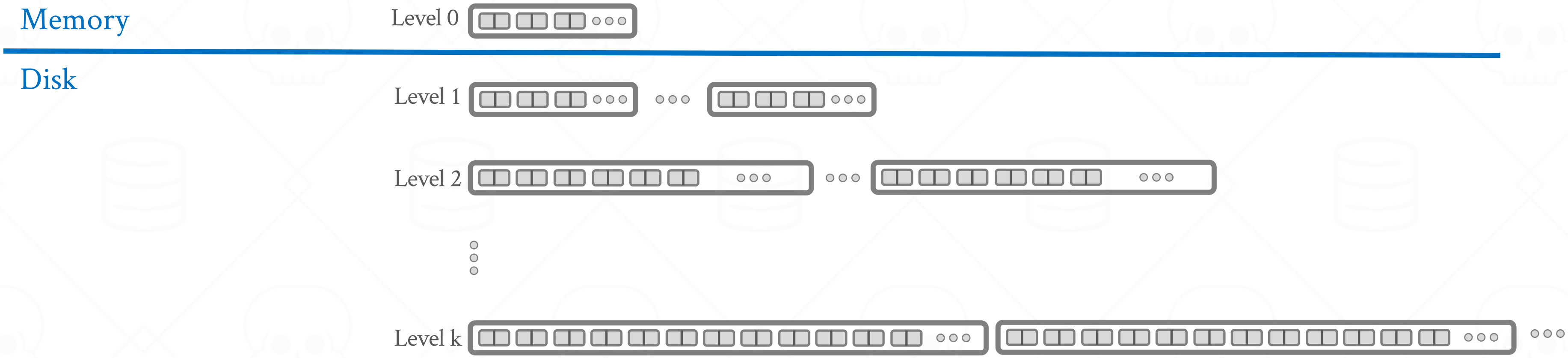


In-memory log: organized a skip list, trie, …
This level is called a MemTable.

A **Level** = sequence of SSTs

Larger SSTs at lower levels.

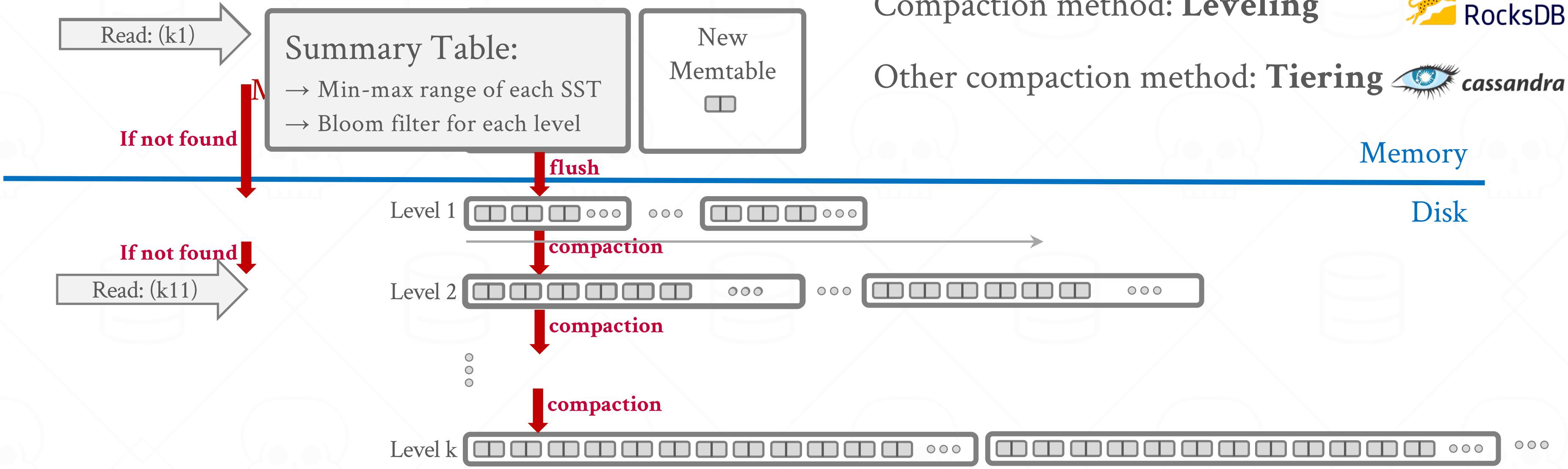
LOG STRUCTURED STORAGE



LOG STRUCTURED STORAGE

Compaction method: **Leveling** RocksDB

Other compaction method: **Tiering** cassandra



What happens if one searches for a key that is not present?

→ May have to search all the levels one-by-one. Expensive!

→ Summary table info can help. Skip the levels that don’t hit in the corresponding bloom filter.

LOG-STRUCTURED STORAGE: DETAILS

Three ops: **GET**, **PUT**, **DELETE**.

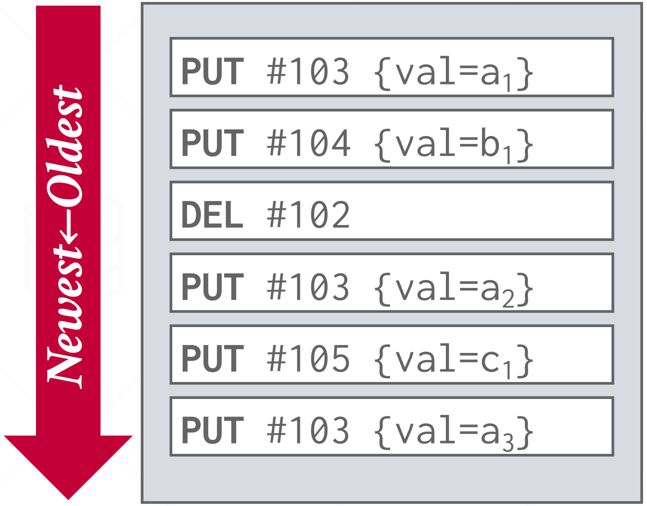
→ **GET**: go through the levels to find the key

For **PUT**and **DELETE** ops:

→ Log changes to a record as appends

→ Each log entry represents a tuple **PUT**/**DELETE** operation.

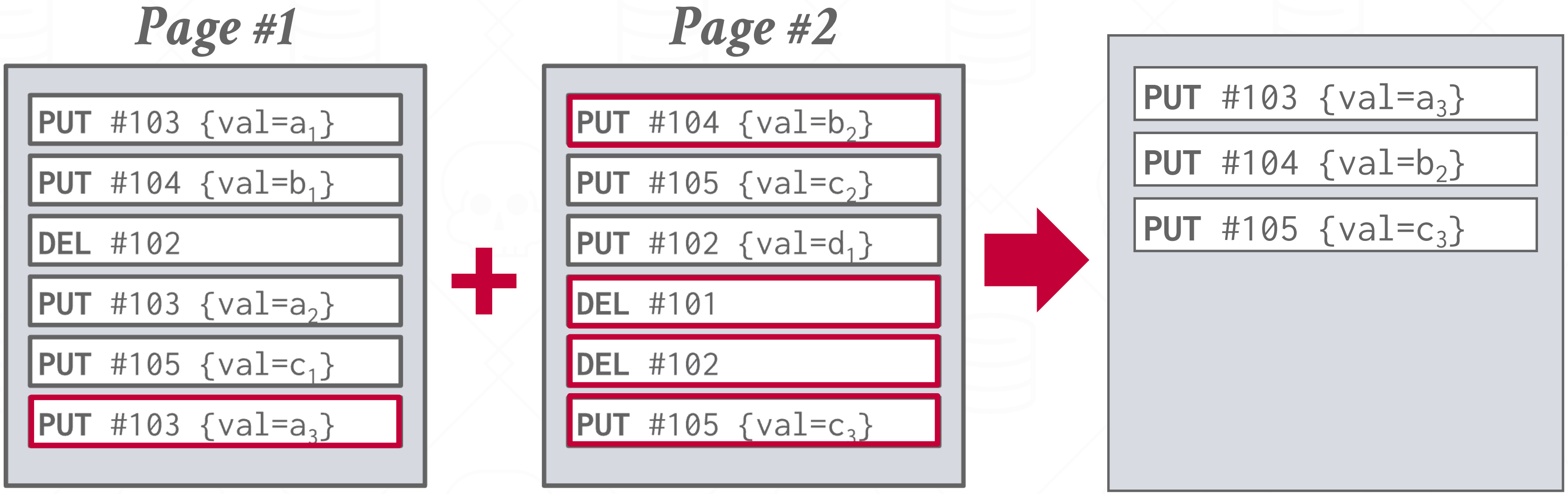
→ **DELETE**s are added as a “tombstone” entry. Not in-place update.



COMPACTION

Compaction = Merge two SSTs to create a larger SST

- → Recall: SSTs are files on disk, so are variable length
- → Uses a sort-merge algorithm
- → Only keep the “latest” values for each key (aka. compacts)



Page #1

PUT #103 {val=a_1}
PUT #104 {val=b_1}
DEL #102
PUT #103 {val=a_2}
PUT #105 {val=c_1}
PUT #103 {val=a_3}

+

Page #2

PUT #104 {val=b_2}
PUT #105 {val=c_2}
PUT #102 {val=d_1}
DEL #101
DEL #102
PUT #105 {val=c_3}

→

PUT #103 {val=a_3}
PUT #104 {val=b_2}
PUT #105 {val=c_3}

COMPACTION ALGORITHMS

Many different ways to do compaction while preserving the overall LSM property

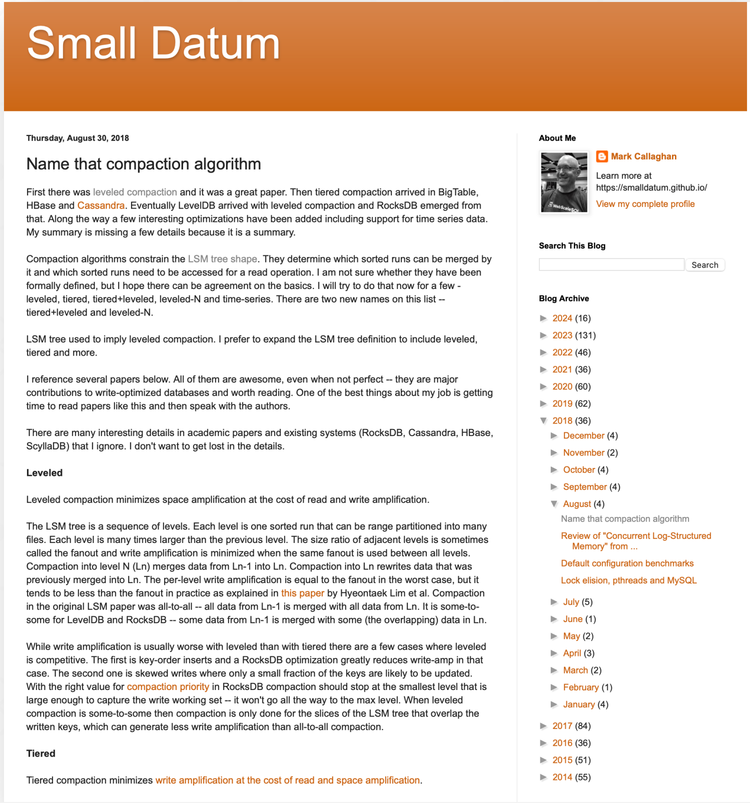
→ Search level-by-level, with newer data at the top levels

Compaction strategies tradeoff:

→ Write amplification

→ Read amplification

→ Space amplification



Small Datum

Thursday, August 30, 2018

Name that compaction algorithm

First there was leveled compaction and it was a great paper. Then tiered compaction arrived in BigTable, HBase and Cassandra. Eventually LevelDB arrived with leveled compaction and RocksDB emerged from that. Along the way a few interesting optimizations have been added including support for time series data. My summary is missing a few details because it is a summary.

Compaction algorithms constrain the LSM tree shape. They determine which sorted runs can be merged by it and which sorted runs need to be accessed for a read operation. I am not sure whether they have been formally defined, but I hope there can be agreement on the basics. I will try to do that now for a few - leveled, tiered, tiered+leveled, leveled-N and time-series. There are two new names on this list -- tiered+leveled and leveled-N.

LSM tree used to imply leveled compaction. I prefer to expand the LSM tree definition to include leveled, tiered and more.

I reference several papers below. All of them are awesome, even when not perfect -- they are major contributions to write-optimized databases and worth reading. One of the best things about my job is getting time to read papers like this and then speak with the authors.

There are many interesting details in academic papers and existing systems (RocksDB, Cassandra, HBase, ScyllaDB) that I ignore. I don't want to get lost in the details.

Leveled

Leveled compaction minimizes space amplification at the cost of read and write amplification.

The LSM tree is a sequence of levels. Each level is one sorted run that can be range partitioned into many files. Each level is many times larger than the previous level. The size ratio of adjacent levels is sometimes called the fanout and write amplification is minimized when the same fanout is used between all levels. Compaction into level N (Ln) merges data from Ln-1 into Ln. Compaction into Ln rewrites data that was previously merged into Ln. The per-level write amplification is equal to the fanout in the worst case, but it tends to be less than the fanout in practice as explained in this paper by Hyeontaek Lim et al. Compaction in the original LSM paper was all-to-all -- all data from Ln-1 is merged with all data from Ln. It is some-to-some for LevelDB and RocksDB -- some data from Ln-1 is merged with some (the overlapping) data in Ln.

While write amplification is usually worse with leveled than with tiered there are a few cases where leveled is competitive. The first is key-order inserts and a RocksDB optimization greatly reduces write-amp in that case. The second one is skewed writes where only a small fraction of the keys are likely to be updated. With the right value for compaction priority in RocksDB compaction should stop at the smallest level that is large enough to capture the write working set -- it won't go all the way to the max level. When leveled compaction is some-to-some then compaction is only done for the slices of the LSM tree that overlap the written keys, which can generate less write amplification than all-to-all compaction.

Tiered

Tiered compaction minimizes write amplification at the cost of read and space amplification.

About Me

Mark Callaghan

Learn more at https://smalldatum.github.io/

View my complete profile

Search This Blog

Search

Blog Archive

- ▶ 2024 (16)
- ▶ 2023 (131)
- ▶ 2022 (46)
- ▶ 2021 (36)
- ▶ 2020 (60)
- ▶ 2019 (62)
- ▼ 2018 (36)
  - ▶ December (4)
  - ▶ November (2)
  - ▶ October (4)
  - ▶ September (4)
  - ▼ August (4)
    - Name that compaction algorithm
    - Review of "Concurrent Log-Structured Memory" from ...
    - Default configuration benchmarks
    - Lock elision, pthreads and MySQL
  - ▶ July (5)
  - ▶ June (1)
  - ▶ May (2)
  - ▶ April (3)
  - ▶ March (2)
  - ▶ February (1)
  - ▶ January (4)
- ▶ 2017 (84)
- ▶ 2016 (36)
- ▶ 2015 (51)
- ▶ 2014 (55)

https://smalldatum.blogspot.com/2018/08/name-that-compaction-algorithm.html

DISCUSSION

Log-structured storage managers are more common today. This is partly due to the proliferation of RocksDB.

What are some downsides of this approach?

→ Reads may be slower.

→ Write amplification.

→ Compaction is expensive.

RocksDB

levelDB

APACHE HBASE

yugabyteDB

fauna

Dgraph

TiDB

CockroachDB

cassandra

OBSERVATION

The two table storage approaches we've discussed so far rely on indexes to find individual tuples.

→ Such indexes are necessary because the tables are inherently unsorted.

But what if the DBMS could keep tuples sorted automatically using an index?

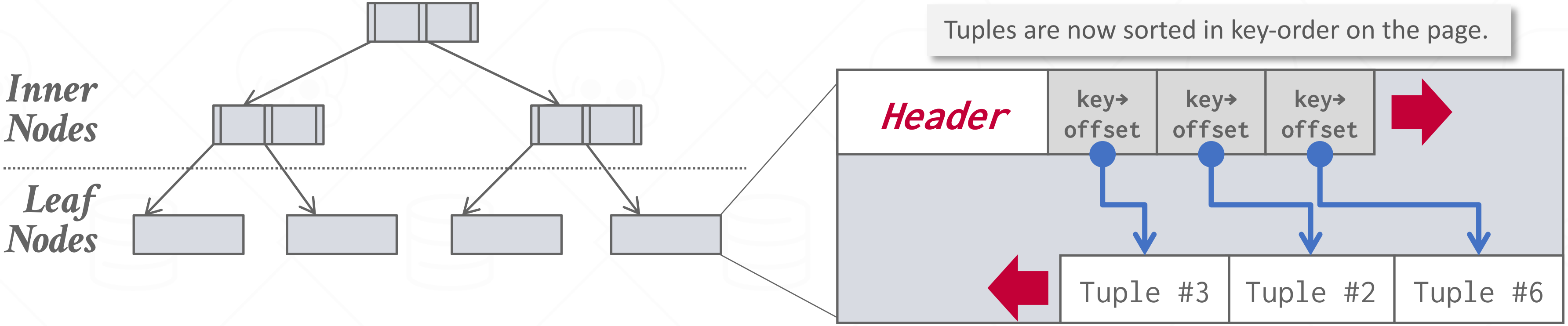
INDEX-ORGANIZED STORAGE

DBMS stores a table's tuples as the value of an index data structure.

→ Still use a page layout that looks like a slotted page.

Tuples are typically sorted in a page based on a key.

SQLite
MySQL
ORACLE
Microsoft SQL Server



Tuples are now sorted in key-order on the page.

TUPLE STORAGE

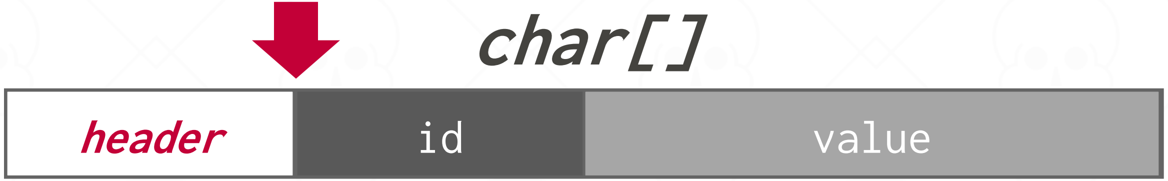
A tuple is essentially a sequence of bytes.

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 AndySux (
  id INT PRIMARY KEY,
  value BIGINT
);
```

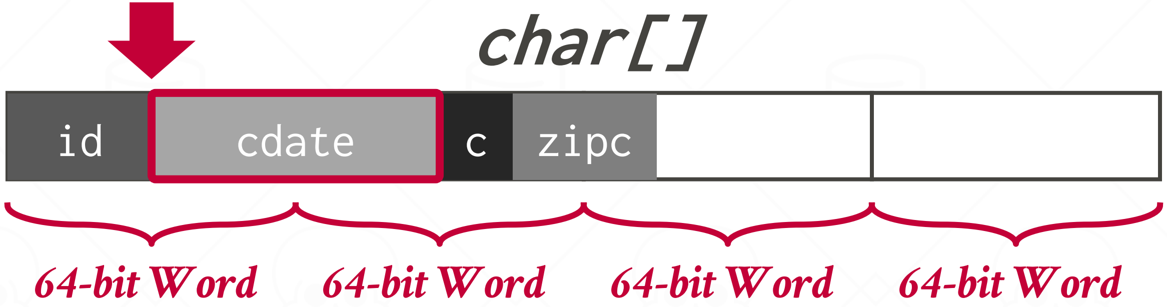


```
reinterpret_cast<int32_t*>(address)
```

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
         );
```



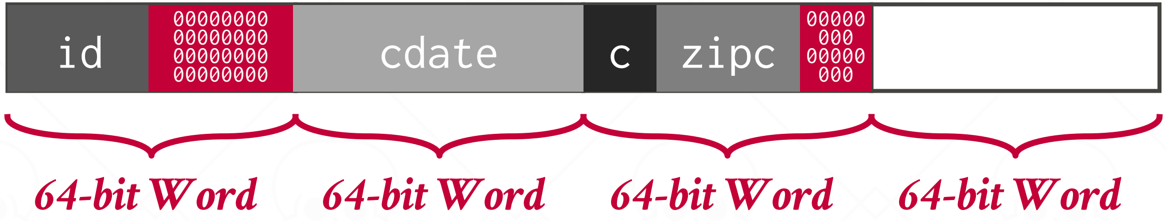
In the old days, the DBMS programmer had to worry about "unaligned word memory reference."

Today: Processors handle it. It will read multiple words from memory, so it may have a performance impact.

WORD-ALIGNMENT: PADDING

Add empty bits after attributes to ensure that tuple is word aligned.

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



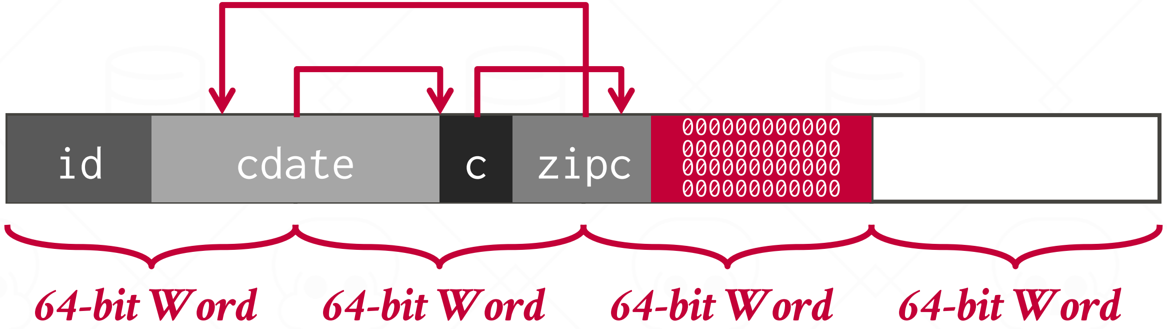
WORD-ALIGNMENT: REORDERING

Switch the order of attributes in the tuples' physical layout to ensure they are aligned.

→ May still have to use padding.

```
CREATE TABLE foo (
  id INT PRIMARY KEY,
  cdate TIMESTAMP,
  color CHAR(2),
  zipcode INT
);
```

32-bits · *64-bits* · *16-bits* · *32-bits*



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>

in

}
```

```
#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
```

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 metadata.

#

Weight of

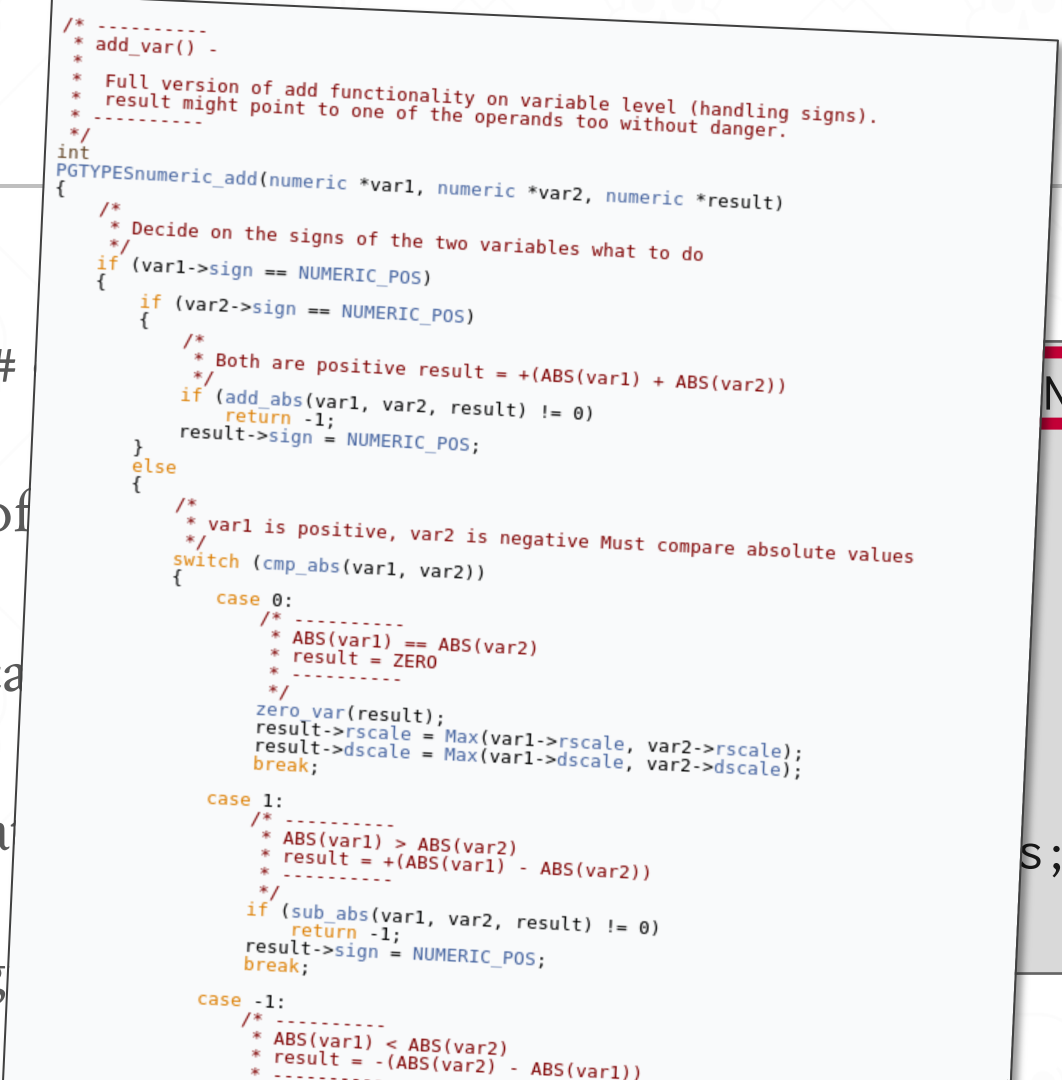
Sca

Positive/Nega

Dig

```
NumericDigit;

s;
```



```
/* ----------
 * add_var() -
 *
 *  Full version of add functionality on variable level (handling signs).
 *  result might point to one of the operands too without danger.
 * ----------
 */
int
PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result)
{
    /*
     * Decide on the signs of the two variables what to do
     */
    if (var1->sign == NUMERIC_POS)
    {
        if (var2->sign == NUMERIC_POS)
        {
            /*
             * Both are positive result = +(ABS(var1) + ABS(var2))
             */
            if (add_abs(var1, var2, result) != 0)
                return -1;
            result->sign = NUMERIC_POS;
        }
        else
        {
            /*
             * var1 is positive, var2 is negative Must compare absolute values
             */
            switch (cmp_abs(var1, var2))
            {
                case 0:
                    /* ----------
                     * ABS(var1) == ABS(var2)
                     * result = ZERO
                     * ----------
                     */
                    zero_var(result);
                    result->rscale = Max(var1->rscale, var2->rscale);
                    result->dscale = Max(var1->dscale, var2->dscale);
                    break;

                case 1:
                    /* ----------
                     * ABS(var1) > ABS(var2)
                     * result = +(ABS(var1) - ABS(var2))
                     * ----------
                     */
                    if (sub_abs(var1, var2, result) != 0)
                        return -1;
                    result->sign = NUMERIC_POS;
                    break;

                case -1:
                    /* ----------
                     * ABS(var1) < ABS(var2)
                     * result = -(ABS(var2) - ABS(var1))
                     * ----------
```

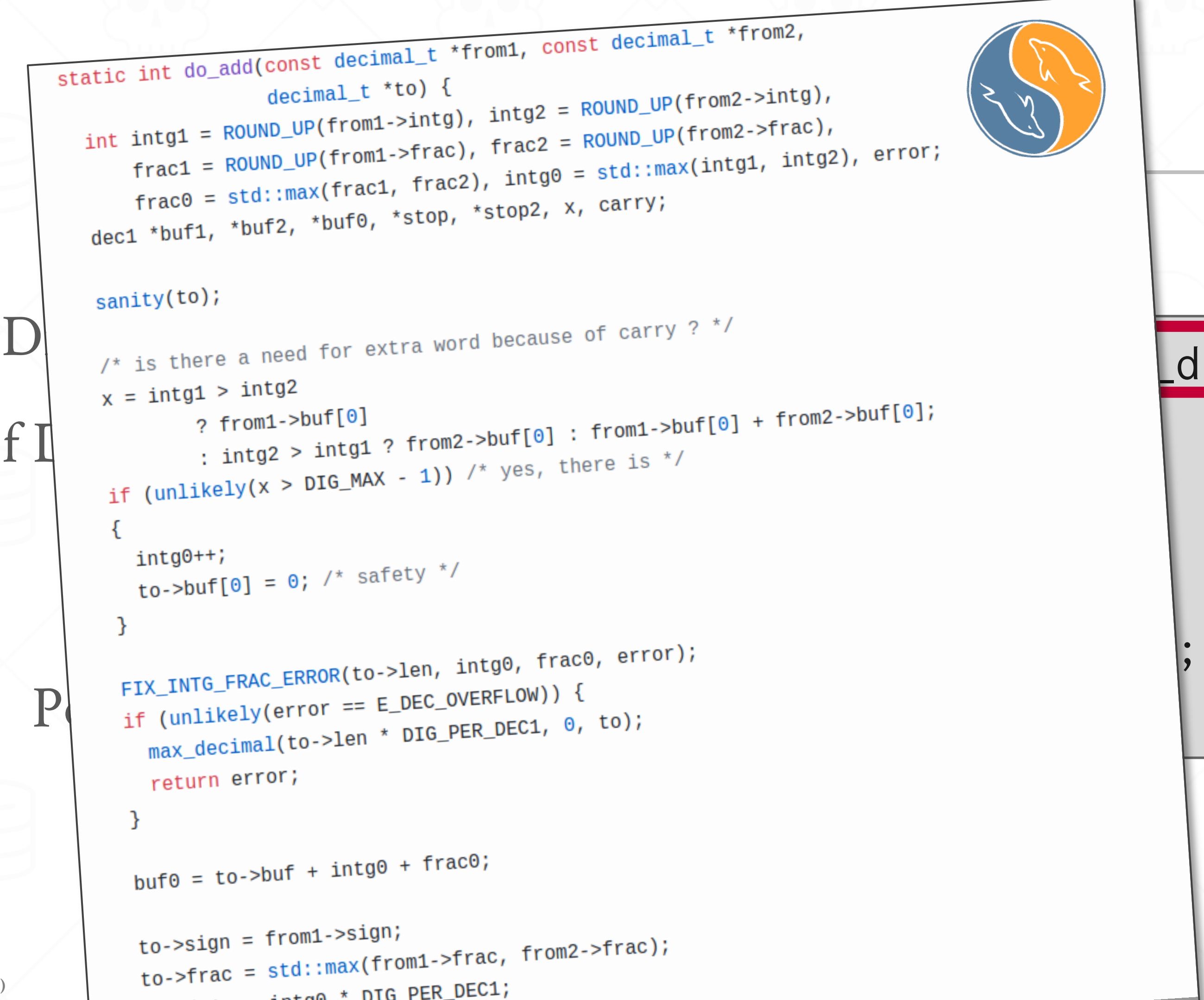
# of D

# of D

P

_digit_t;

;



```
static int do_add(const decimal_t *from1, const decimal_t *from2,
                  decimal_t *to) {
  int intg1 = ROUND_UP(from1->intg), intg2 = ROUND_UP(from2->intg),
      frac1 = ROUND_UP(from1->frac), frac2 = ROUND_UP(from2->frac),
      frac0 = std::max(frac1, frac2), intg0 = std::max(intg1, intg2), error;
  dec1 *buf1, *buf2, *buf0, *stop, *stop2, x, carry;

  sanity(to);

  /* is there a need for extra word because of carry ? */
  x = intg1 > intg2
          ? from1->buf[0]
          : intg2 > intg1 ? from2->buf[0] : from1->buf[0] + from2->buf[0];
  if (unlikely(x > DIG_MAX - 1)) /* yes, there is */
  {
    intg0++;
    to->buf[0] = 0; /* safety */
  }

  FIX_INTG_FRAC_ERROR(to->len, intg0, frac0, error);
  if (unlikely(error == E_DEC_OVERFLOW)) {
    max_decimal(to->len * DIG_PER_DEC1, 0, to);
    return error;
  }

  buf0 = to->buf + intg0 + frac0;

  to->sign = from1->sign;
  to->frac = std::max(from1->frac, from2->frac);
                intg0 * DIG_PER_DEC1;
```

NULL DATA TYPES

Choice #1: Null Column Bitmap Header

→ Store a bitmap in a centralized header that specifies what attributes are null.

This is the most common approach.

Choice #2: Special Values

→ Designate a value to represent `NULL` for a data type (e.g., `INT32_MIN`).

Choice #3: Per Attribute Null Flag

→ Store a flag that marks that a value is null.

→ Must use more space than just a single bit because this messes up with word alignment.

LARGE VALUES

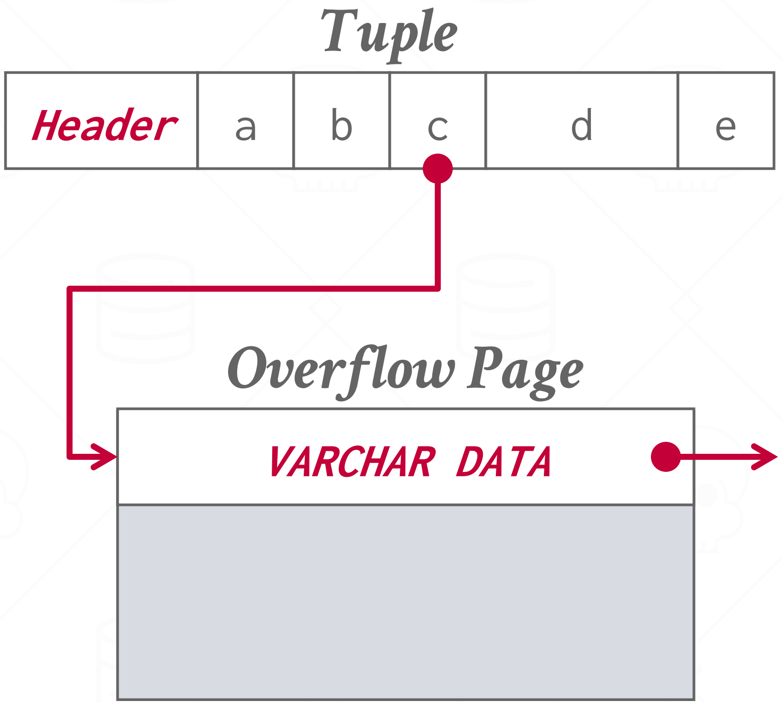
Most DBMSs don't allow a tuple to exceed the size of a single page.

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)



EXTERNAL VALUE

Some systems allow you to store a large value in an external file.
Treated as a **BLOB** type.

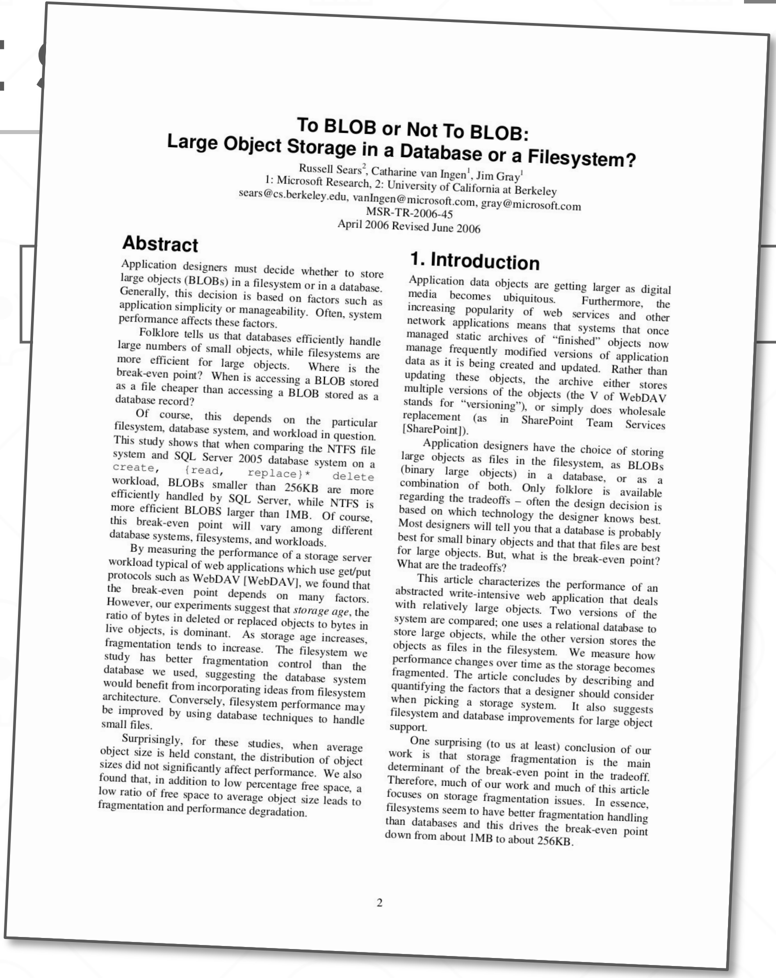
→ Oracle: **BFILE** data type

→ 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 Sears[2], Catharine van Ingen[1], Jim Gray[1]
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.

2

SYSTEM CATALOGS

A DBMS stores meta-data about databases in its internal catalogs.

→ Tables, columns, indexes, views

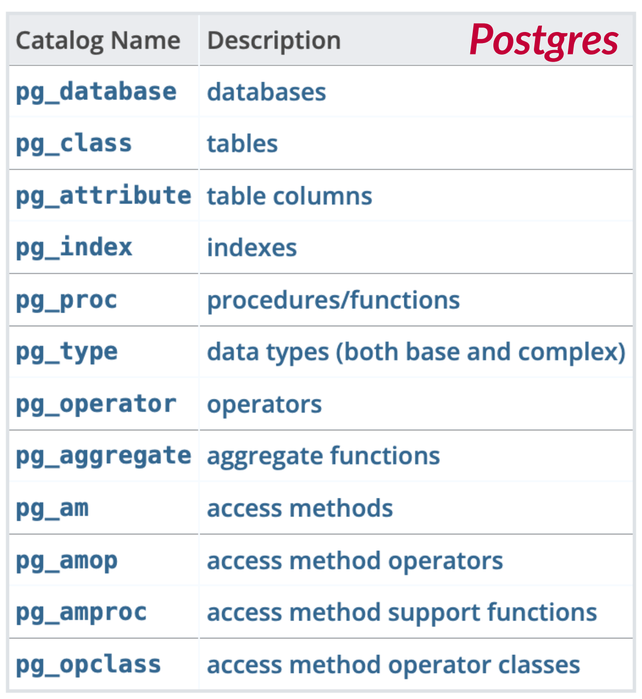
→ Users, permissions

→ Internal statistics

Almost every DBMS stores the database's catalog inside itself (i.e., as tables).

→ Wrap object abstraction around tuples.

→ Specialized code for "bootstrapping" catalog tables.



Postgres

Catalog Name	Description
pg_database	databases
pg_class	tables
pg_attribute	table columns
pg_index	indexes
pg_proc	procedures/functions
pg_type	data types (both base and complex)
pg_operator	operators
pg_aggregate	aggregate functions
pg_am	access methods
pg_amop	access method operators
pg_amproc	access method support functions
pg_opclass	access method operator classes

SYSTEM CATALOGS

You can query the DBMS's internal **INFORMATION_SCHEMA** catalog to get info about the database.

→ ANSI standard set of read-only views that provide info about all the tables, views, columns, and procedures in a database

DBMSs also have non-standard shortcuts to retrieve this information.

ACCESSING TABLE SCHEMA

List all the tables in the current database:

SQL-92

```
SELECT *
  FROM INFORMATION_SCHEMA.TABLES
 WHERE table_catalog = '<db name>';
```

Postgres

```
\d;
```

MySQL

```
SHOW TABLES;
```

SQLite

```
.tables
```

ACCESSING TABLE SCHEMA

List all the tables in the student table:

SQL-92

```
SELECT *
  FROM INFORMATION_SCHEMA.TABLES
 WHERE table_name = 'student'
```

Postgres

```
\d student;
```

MySQL

```
DESCRIBE student;
```

SQLite

```
.schema student
```

INDEXES

CREATE INDEX:

→ Scan the entire table and populate the index.

→ Have to record changes made by txns that modified the table while another txn was building the index.

→ When the scan completes, lock the table and resolve changes that were missed after the scan started.

DROP INDEX:

→ Just drop the index logically from the catalog.

→ It only becomes "invisible" when the txn that dropped it commits. All existing txns will still have to update it.

CONCLUSION

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

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

The storage manager is not entirely independent from the rest of the DBMS.