 Intro to Database Systems (15-445/645)

04 Database Storage

Part 2

Carnegie Mellon University

SPRING 2023

Charlie Garrod
ADMINISTRIVIA

Project 0 due yesterday.

Homework 1 due this Friday, Feb 3rd.

Project 1 will be released later today.
DISK-ORIENTED ARCHITECTURE

The DBMS assumes that the primary storage location of the database is on non-volatile disk.

The DBMS's components manage the movement of data between non-volatile and volatile storage.
PAGE-ORIENTED ARCHITECTURE

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.
→ Overwrite existing data (if new data fits).
DISCUSSION

Problems with the slotted page design
→ Fragmentation
→ Useless Disk I/O
→ Random Disk I/O (e.g., update 20 tuples on 20 pages)

What if the DBMS cannot overwrite data in pages and could only create new pages?
→ Examples: Some cloud storage, HDFS
TODAY'S AGENDA

Log-Structured Storage
Data Representation
System Catalogs
DBMS stores log records that contain 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 STORAGE

When the page gets full, the DBMS writes it out disk and starts filling up the next page with records.
→ All disk writes are sequential.
→ On-disk pages are immutable.

In-Memory Page

PUT #104 {val=b_2}
PUT #105 {val=c_2}
PUT #102 {val=d_1}
DEL #101
DEL #102
PUT #105 {val=c_3}
LOG-STRUCTURED STORAGE

To read a tuple with a given id, the DBMS finds the newest log record corresponding to that id.
→ Scan log from newest to oldest.

Maintain an index that maps a tuple id to the newest log record.
→ If log record is in-memory, just read it.
→ If log record is on a disk page, retrieve it.
→ We will discuss indexes in two weeks.
The log will grow forever. The DBMS needs to periodically compact pages to reduce wasted space.

Page 1
- PUT #103 {val=a₁}
- PUT #104 {val=b₁}
- DEL #102
- PUT #103 {val=a₂}
- PUT #105 {val=c₁}
- PUT #103 {val=a₃}

Page 2
- PUT #104 {val=b₂}
- PUT #105 {val=c₂}
- PUT #102 {val=d₁}
- DEL #101
- DEL #102
- PUT #105 {val=c₃}

PUT #103 {val=a₃}
PUT #104 {val=b₂}
PUT #105 {val=c₃}
DEL #101
DEL #102
After a page is compacted, the DBMS does not need to maintain temporal ordering of records within the page.
→ Each tuple id is guaranteed to appear at most once in the page.

The DBMS can instead sort the page based on id order to improve efficiency of future look-ups.
→ Called Sorted String Tables (SSTables)
LOG-STRUCTURED COMPACTION

Compaction coalesces larger log files into smaller files by removing unnecessary records.

Universal Compaction

Level 0
Level 1
Level 2

Sorted Log File
Sorted Log File
Sorted Log File

Sorted Log File
Sorted Log File
Sorted Log File

Sorted Log File
Sorted Log File

Sorted Log File

Level Compaction

Level 0
Level 1
Level 2

Sorted Log File
Sorted Log File

Sorted Log File

Sorted Log File
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?
→ Write-Amplification
→ Compaction is Expensive
TUPLE STORAGE

A tuple is essentially a sequence of bytes. It's 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 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.
→ Need to worry about collations / sorting.

TIME/DATA/TIMESTAMP
→ 32/64-bit integer of (micro)seconds since Unix epoch
VARIABLE PRECISION NUMBERS

Inexact, variable-precision numeric type that uses the "native" C/C++ types.
→ Examples: FLOAT, REAL/DDOUBLE

Store directly as specified by IEEE-754.

Typically faster than arbitrary precision numbers but can have rounding errors…
# VARIABLE PRECISION NUMBERS

## Rounding Example

```c
#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 = 0.30000001192092895508
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 you give up arbitrary precision.
typedef unsigned char NumericDigit;

typedef struct {
    int ndigits;
    int weight;
    int scale;
    int sign;
    NumericDigit *digits;
} numeric;

int PGTYPES_numeric_add(numeric *var1, numeric *var2, numeric *result)
{
    /* Decide on the signs of the two variables what to do */
    if (var1->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
                 */
                /* zeroiVar(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))
                 */
                break;
            default:
                break;
        }
    }
    return 0;
}
typedef int32 decimal_digit_t;

struct decimal_t {
    int intg, frac, len;
    bool sign;
    decimal_digit_t *buf, *buf0, *stop, *stop2, x, carry;
}

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);
    ...; /* intg DEC1;
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)
Some systems allow you to store a really 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.
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.
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:

```
SELECT * FROM INFORMATION_SCHEMA.TABLES
WHERE table_catalog = '<db name>';  # SQL-92
```

- Postgres: `
  \d;`
- MySQL: `SHOW TABLES;`
- SQLite: `.tables`
ACCESSING TABLE SCHEMA

List all the tables in the student table:

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

- SQL-92
- \d student; **Postgres**
- DESCRIBE student; **MySQL**
- .schema student **SQLite**
CONCLUSION

Log-structured storage is an alternative approach to the page-oriented architecture we discussed last class.

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