Lecture #04
Database Storage Part 2
Homework #1 is due February 2\textsuperscript{nd} @ 11:59pm.

Project #1 is due February 18\textsuperscript{th} @ 11:59pm.

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

Memory

Disk

Level 0

Level 1

Level 2

Level k
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.
→ DELETEs are added as a “tombstone” entry. Not in-place update.
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=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}
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
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.
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.
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.
CREATE TABLE AndySux (
  id INT PRIMARY KEY,
  value BIGINT
);

char[]

header | id | value

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 ( id INT PRIMARY KEY, cdate TIMESTAMP, color CHAR(2), 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.

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

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

```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 metadata.
/* add_var() -
* Full version of add functionality on variable level (handling signs).
* Result might point to one or 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:
          /* result = ZERO */
        
          /* zero var(result); */
          result->ndigits = Max(var1->ndigits, var2->ndigits);
          result->weight = Max(var1->weight, var2->weight);
          break;
        case 1:
          /* result = + (ABS(var1) - ABS(var2)) */
        
          /* if (SUB_ABS(var1, var2, result) != 0) */
          if (sub_abs(var1, var2, result) != 0)
            return -1;
          result->sign = NUMERIC_POS;
          break;
        case -1:
          /* result = - (ABS(var2) - ABS(var1)) */
        
          if (sub_abs(var1, var2, result) != 0)
            return -1;
          result->sign = NUMERIC_NEG;
          break;
        case 2:
          /* result = - ABS(var1) */
        
          if (sub_abs(var1, var2, result) != 0)
            return -1;
          result->sign = NUMERIC_NEG;
          break;
        case -2:
          /* result = ABS(var2) */
        
          if (sub_abs(var1, var2, result) != 0)
            return -1;
          result->sign = NUMERIC_POS;
          break;
      }
    }
  }
}
typedef int32 decimal_digit_t;
struct decimal_t {
    int intg, frac, len;
    bool sign;
    decimal_digit_t *buf;
};
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.
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 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.
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.
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>';
```

- 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
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.
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.