CS 455
Principles of Database Systems

UNIVERSITY of PUGET SOUND
Est. 1888

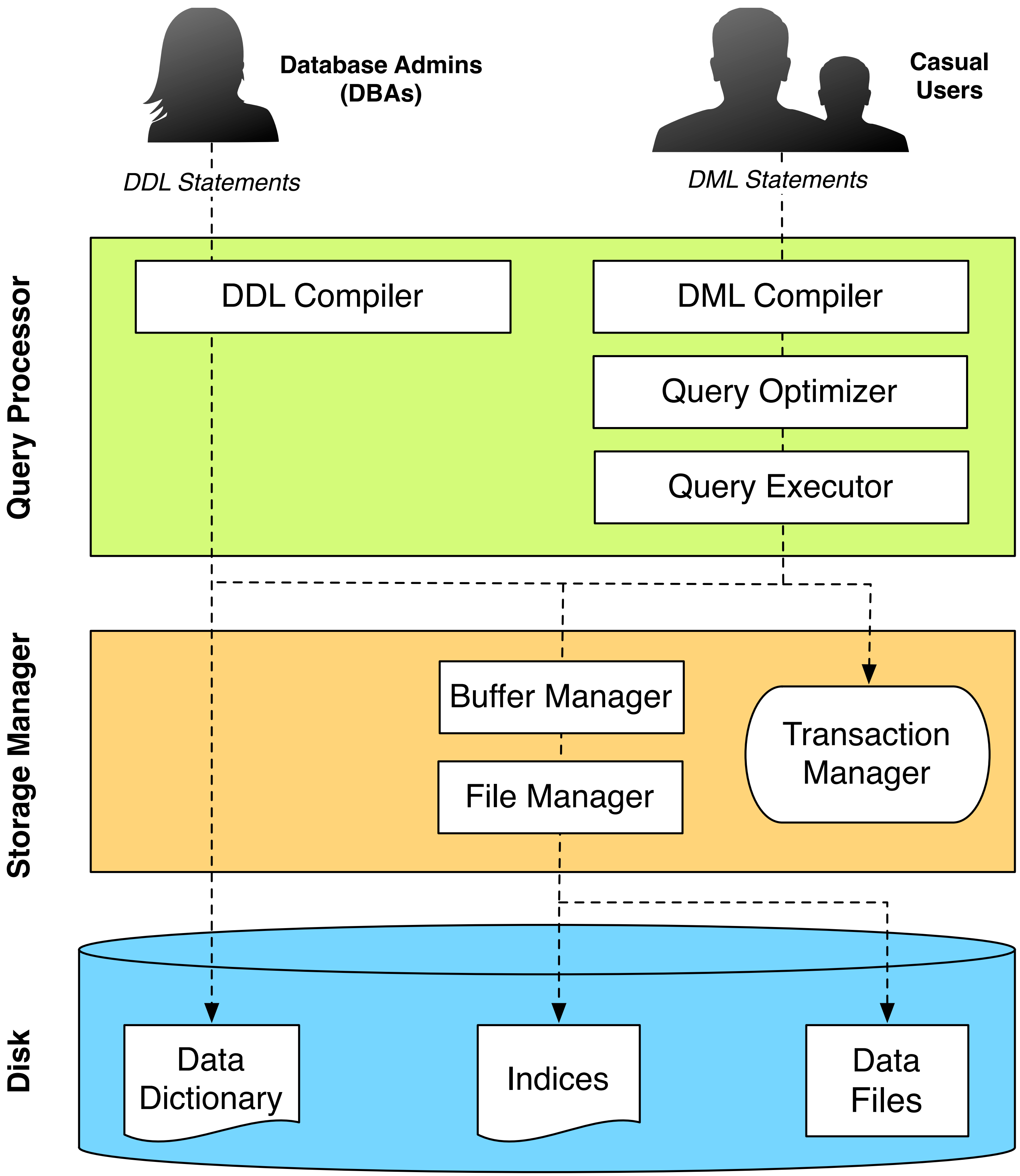
Department of Mathematics
and Computer Science

Lecture6
File Organization, Cost Analysis

Topics

- Database File Structure
  - Fixed-Length Tuples
  - Variable-Length Tuples
- Tuple Organization in Files
  - Heap, Sorted, Hashed
- Cost Analysis
- Join Processing
  - Nested-Loop Join (NLJ)
  - Hash Join (HJ)
  - Sort-Merge Join (SMJ)

Database Architecture



Data Dictionary
Stores metadata about the schema of the database.

Indices
Data structures that provide fast-access to data.

Data Files
Stores the database itself.

Storing Tuples: What If DBs *Know* Tuple Sizes?

- Fixed length attributes and tuples
- Consider this DDL statement in MySQL
  - The parenthesized integer is a limit on length

```
CREATE TABLE faculty (
    facID    VARCHAR(5),
    lastname VARCHAR(20),
    dept     VARCHAR(10),
    salary   INT(4),
    PRIMARY KEY(facID)
);
```

tuple	facID	lastname	dept	salary
byte	1 — 5	6 — 25	26 — 35	36 — 40

DB Dictionary

Name: Faculty

Attribute	Start	Len
facID	1	5
lastname	6	20
dept	26	10
salary	36	4

Name: ...

Storing Tuples: Fixed-Length Approach

- Fixed-Length is the simplest way to store tuples:
  - Store tuple *t* starting at byte *n*t*, where *n* is the size of each tuple

0	10101	Srinivasan	Comp. Sci.	65000	40 Bytes (B)
1	12121	Wu	Finance	90000	40 B
2	15151	Mozart	Music	40000	40 B
3	22222	Einstein	Physics	95000	40 B
4	32343	El Said	History	60000	40 B
5	33456	Gold	Physics	87000	40 B
6	45565	Katz	Comp. Sci.	75000	40 B
7	58583	Califieri	History	62000	40 B
8	76543	Singh	Finance	80000	40 B
9	76766	Crick	Biology	72000	40 B
10	83821	Brandt	Comp. Sci.	92000	40 B
11	98345	Kim	Elec. Eng.	80000	40 B

Query Processing

- Process this query: `SELECT * FROM faculty WHERE C;`
  - Assume:
    - The data dictionary tells us that the size of a tuple: `SIZE = 40` bytes
  - Tuple extraction is super fast! Implementation below:

```
public Set<Tuple> select(Query q, Relation table, Condition cond) {
    Set<Tuple> result = new HashSet<>();

    FileInputStream file = new FileInputStream(table);
    byte[] row = new byte[SIZE];  //40 is the tuple size in the table

    while (file.read(row, SIZE) != -1) //hasn't reached EOF
        if (q.evaluate(row, cond))
            result.add(new Tuple(row));

    file.close();
    return result;
}
```

Storing Tuples: Fixed-Length Approach

- Projection is also fast when sizes of each attribute are known!

0	10101	Srinivasan	Comp. Sci.	65000
1	12121	Wu	Finance	90000
2	15151	Mozart	Music	40000
3	22222	Einstein	Physics	95000
4	32343	El Said	History	60000
5	33456	Gold	Physics	87000
6	45565	Katz	Comp. Sci.	75000
7	58583	Califieri	History	62000
8	76543	Singh	Finance	80000
9	76766	Crick	Biology	72000
10	83821	Brandt	Comp. Sci.	92000
11	98345	Kim	Elec. Eng.	80000

DB Dictionary

Name: Faculty

Attribute	Start	Len
facID	1	5
lastname	6	20
dept	26	10
salary	36	4

tuple	facID	lastname	dept	salary
byte	1 — 5	6 — 25	26 — 35	36 — 40

Storing Tuples: Fixed-Length Approach (Cont.)

- Problem:
  - What do we do when tuples are deleted?

Possible Solutions:

- Opt 1: Compact the block immediately so free space is always toward the end.
- Opt 2: Keep track of the positions of deleted tuples. Don't use that in the future.
- Opt 3: Manage a *"free list"*
  (Opt 3 is done in practice)

0	10101	Srinivasan	Comp. Sci.	65000
1	~~12121~~	~~Wu~~	~~Finance~~	~~90000~~
2	15151	Mozart	Music	40000
3	22222	Einstein	Physics	95000
4	~~32343~~	~~El Said~~	~~History~~	~~60000~~
5	33456	Gold	Physics	87000
6	~~45565~~	~~Katz~~	~~Comp. Sci.~~	~~75000~~
7	58583	Califieri	History	62000
8	76543	Singh	Finance	80000
9	76766	Crick	Biology	72000
10	83821	Brandt	Comp. Sci.	92000
11	98345	Kim	Elec. Eng.	80000

Storing Tuples: Fixed-Length Approach (Cont.)

- Free Lists for organizing deleted tuples
  - Store the address of the first deleted record in a *file header*
    - Use the first deleted record to store address of the second deleted record, etc.

Add this to top of each relation's file

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Fixed-Length Tuples Summary

- Pros:
  - Tuple extraction and attribute projection are **super fast**!
  - *Free Lists* are also efficient and easy to implement
    - File header is an negligible space overhead

- Cons:
  - Not very flexible for all kinds of data
    - We often don’t know how long a tuple (or certain attributes) might be
    - Don't want to limit users on length so we tend to "over-provision"

Topics

- Database File Structure
  - Fixed-Length Tuples
  - Variable-Length Tuples
- Tuple Organization in Files
  - Heap, Sorted, Hashed
- Cost Analysis
- Join Processing
  - Nested-Loop Join (NLJ)
  - Hash Join (HJ)
  - Sort-Merge Join (SMJ)

What If We *Don't* Know the Tuple Size?

- Consider this DDL statement in SQLite

```sql
CREATE TABLE faculty (
    facID    TEXT,
    lastname TEXT,
    dept     TEXT,
    salary   INTEGER,
    PRIMARY KEY(facID)
);
```

- *F, L, D, S byte offsets* below may vary for each tuple in the Faculty table

faculty tuple	facID	lastname	dept	salary
byte	1 — F	F+1 — L	L+1 — D	D+1 — S

Variable-Length Tuples

- Problems:
  - We no longer know where each attribute starts and ends...
  - And, how do we project certain attributes?

	facID	lastname	dept	salary
	'1102'	'Wallace'	'CS'	45000
byte	1 — 4	5 — 11	12 13	14 — 17

	facID	lastname	dept	salary
	'89'	'Srinivasan'	'English'	50000
byte	1 – 2	3 — 12	13 — 19	20 — 23

Projecting Attributes in Var-Length Tuples

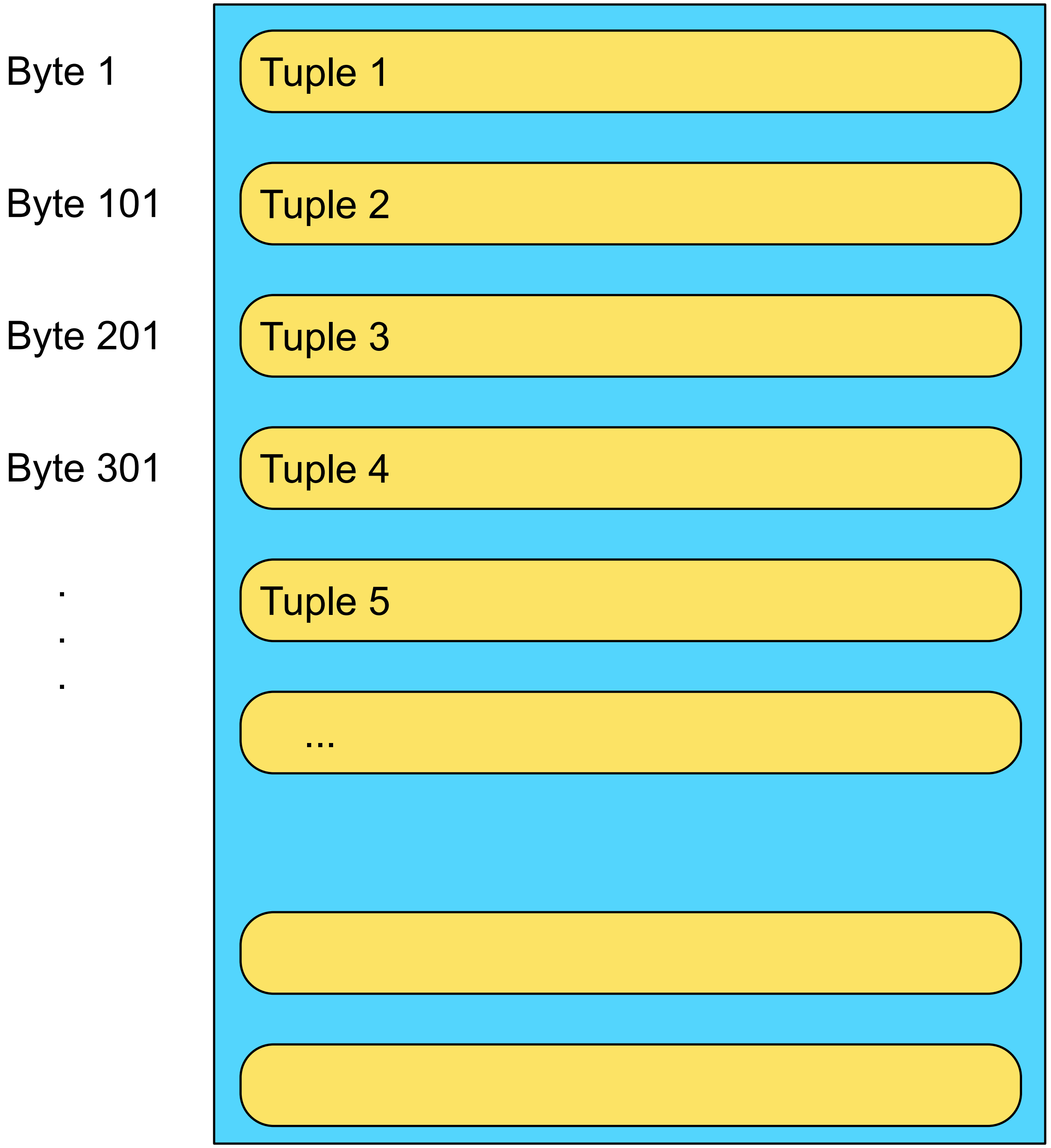
- **Solution:** For each tuple, store a *tuple header* that tells us the *(offset,size)* of its attribute values!
  - Add some metadata to the beginning of *each* tuple

Header	facID	lastname	dept	salary
(17,4)(22,7)(29,2)(31,4)	'1102'	'Wallace'	'CS'	55000
1 — 16	17 — 20	21 — 28	29 30	31 — 34

Header	facID	lastname	dept	salary
(17,2)(19,10)(29,7)(37,4)	'89'	'Srinivasan'	'English'	50000
1 — 16	17 — 18	19 — 28	29 — 36	37 — 31

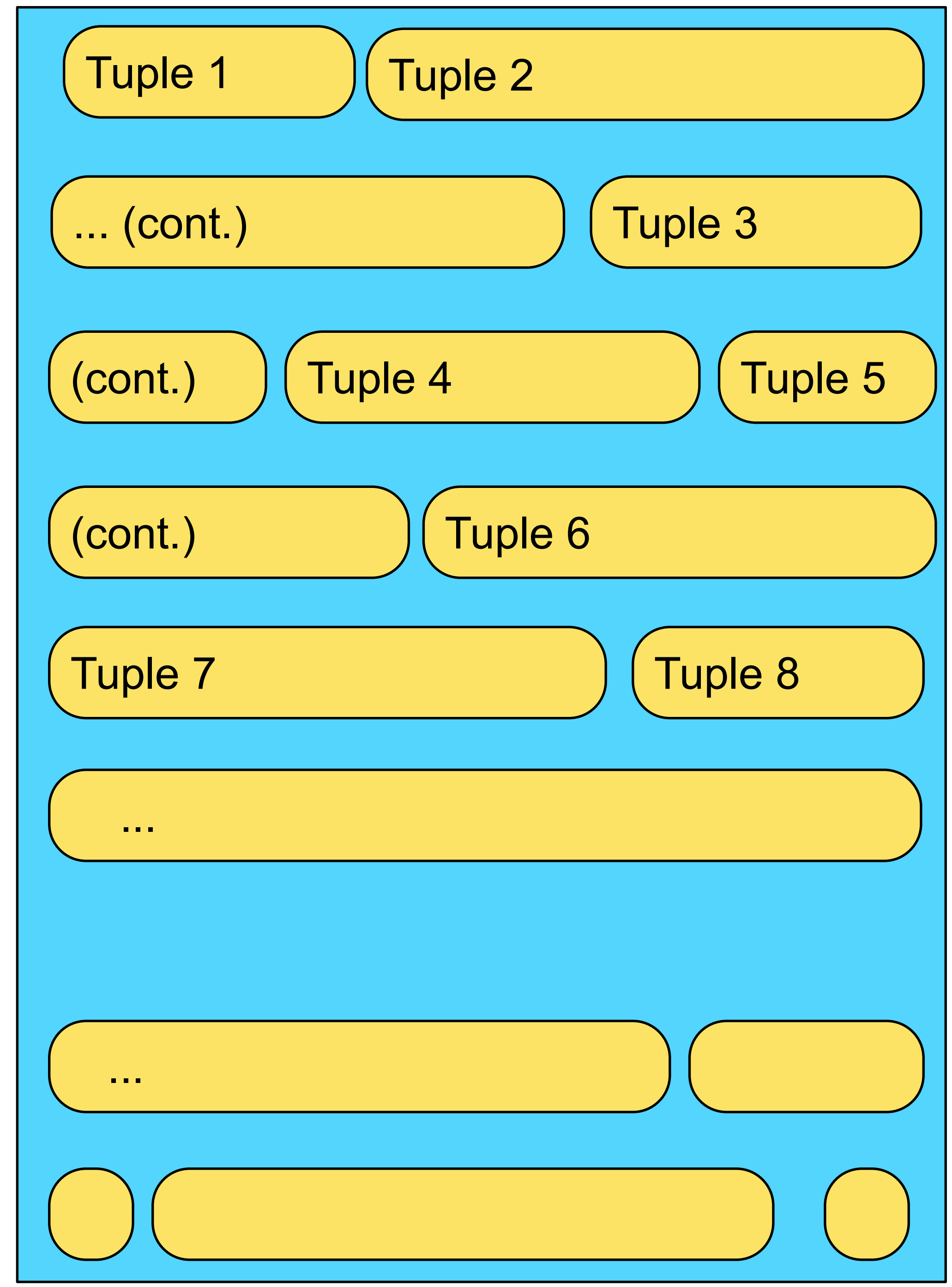
Extracting Variable-Length Tuples

If a tuple is fixed at 100 B,



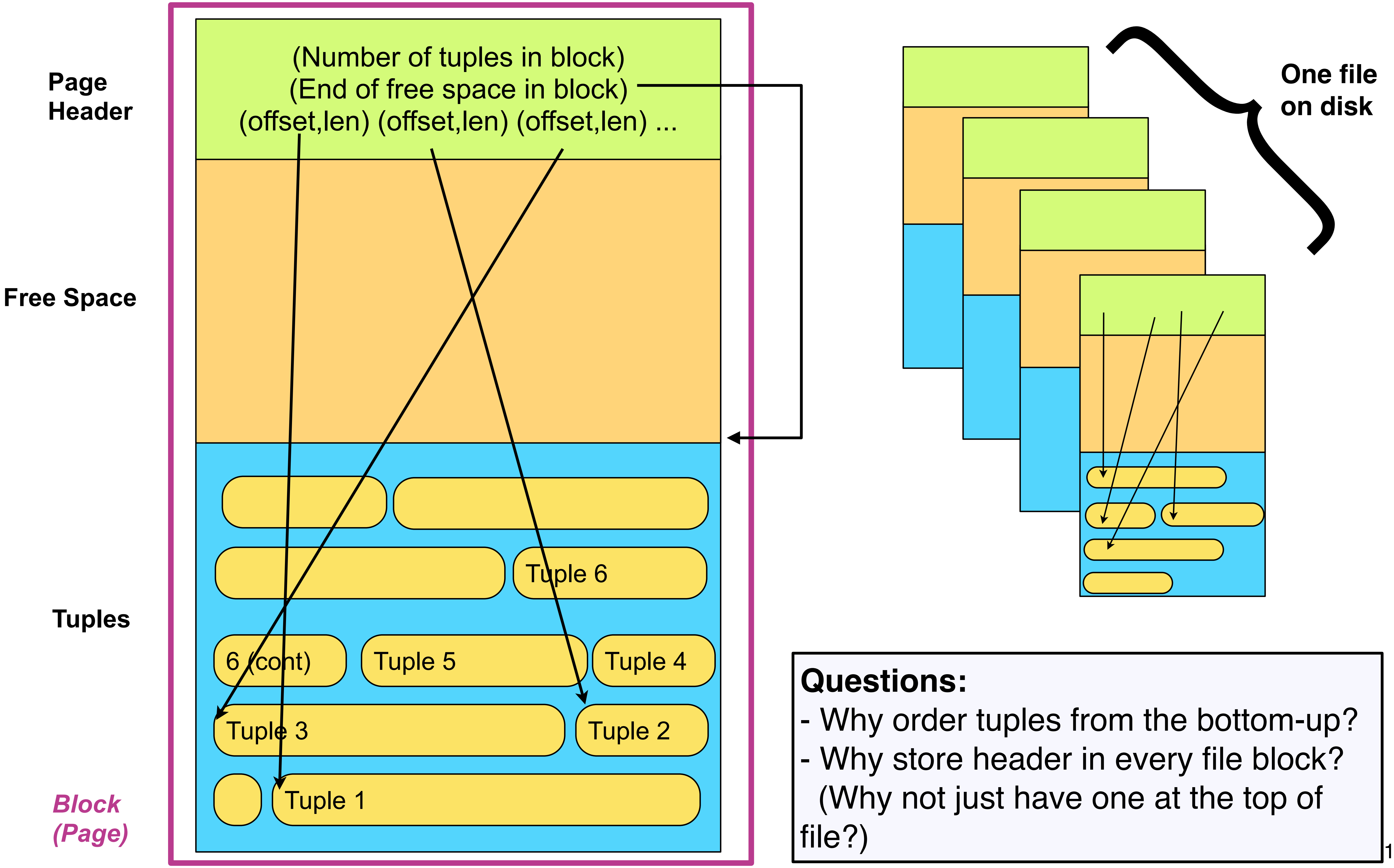
Fixed-Length Tuples
Easy to compute where a tuple starts/ends
We can extract tuples easily

If a tuple is variable length,



Variable-Length Tuples
Now we have *no idea* where each starts/ends...

Slotted-Pages for Storing Variable-Len Tuples



Questions:
- Why order tuples from the bottom-up?
- Why store header in every file block?
  (Why not just have one at the top of file?)

Variable-Length Tuples (Cont.)

- Pros:
  - Tuples can now be dynamically sized! Not throwing space away!
- Cons:
  - What does it mean for the performance of:
    - Tuple extraction? Attribute projection?
  - Space overhead can be significant for large tables
    - Every tuple has a (4*N)-byte header, where N = number of attributes

Tuple Header	empID	name	cartID	title	wage
(21,1)(22,3)(25,3)(28,8)(36,4)	'0'	'Tim'	'123'	'Fry Cook'	15
1 20	21	22 24	25 27	28 35	36 39

Topics

- Database File Structure
  - Fixed-Length Tuples
  - Variable-Length Tuples
- Tuple Organization
  - Heap, Sorted, Hashed
- Cost Analysis
- Join Processing
  - Nested-Loop Join (NLJ)
  - Hash Join (HJ)
  - Sort-Merge Join (SMJ)

Option 1: Heap File Organization

- *Heap File Organization*
  - Not at all like a "binary heap" that you learned in CS 2
  - Exploits data independence provided by set theory
    - No ordering imposed on tuples
  - A tuple can be placed *anywhere* in the file

Heap Files (Cont.)

- Best when the typical operation on the relation must scan all tuples.

```sql
-- full scan
select * from R;

-- aggregation
select MAX(salary), MIN(salary), AVG(salary) from employee;

-- cartesian product
select * from employee, companies;
```

- Heap files are also great for *insert-heavy workloads (why?)*

Heap File Summary

- Pros:
  - Scan-type queries
    - e.g., Select-All queries and aggregation queries <u>without</u> WHERE clauses
    - Scan queries *are* quite common in real life
  - Great for workloads with lots of insertions (businesses, sciences)!
- Cons:
  - Scans and insertions probably don't dominate an enterprise's workload
  - Equality and range queries, deletions, and updates all suffer (Why?)

Option 2: Sorted Files

- *Sorted File Organization*
  - Tuples are stored in sorted order on the value of a chosen *search key*
    - Not necessarily (but can be) the primary

- Best when:
  - When only a range of tuples are needed

```
SELECT * FROM employee WHERE salary > 80000;
```

  - Exact-match queries (so-so performance; better than heap)

```
SELECT * FROM employee WHERE lastname='Johnson';
```

Option 2: Sorted Files (Cont.)

- For instance, a job-application DB for a restaurant:
  - ssn is the primary key, but HR rarely looks applicants up by ssn
  - Instead, the restaurant is more interested in the applicants' *age*
    - Under 18 is un-hirable
    - 18+ can wait and host
    - 21+ can also bartend

```
CREATE TABLE applicants (
    ssn  TEXT primary key,
    name TEXT,
    age  INT);
```

- Most queries to this table search on an age range. Then sort tuples in file by age.

```
SELECT name AS CanBartend
FROM applicants
WHERE age >= 21
ORDER BY age;
```

Example

applicants

- 122-03-9734, Lisa, **9**
- 111-11-1111, Gabriel, **21**
- 321-49-2832, Bart, **10**
- **...**
- 222-22-2222, Homer, **50**
- 373-32-2111, Joshua, **40**
- 222-49-2832, Ryan, **20**

Heap File

applicants

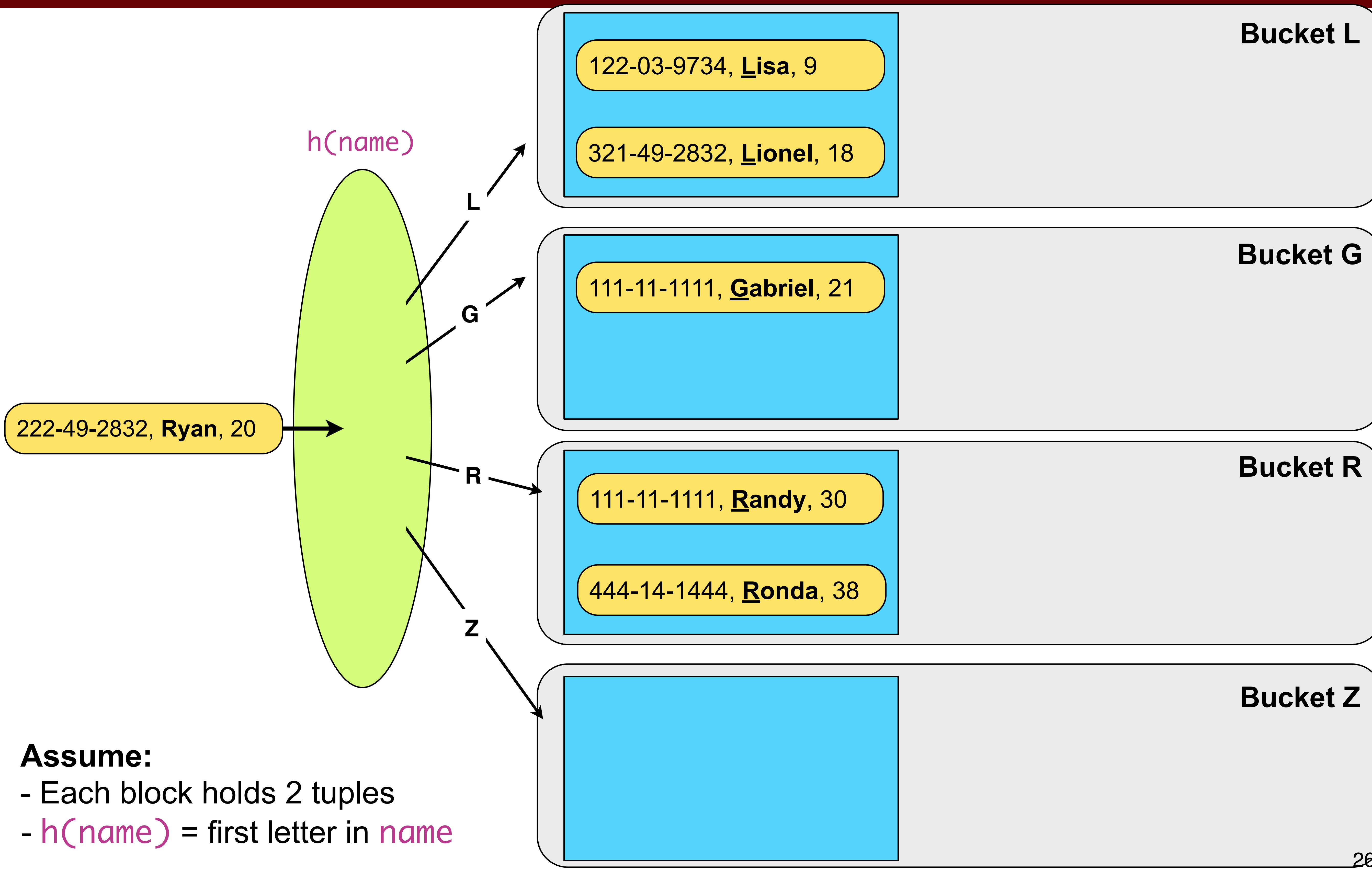
- 122-03-9734, Lisa, **9**
- 321-49-2832, Bart, **10**
- **...**
- 222-49-2832, Ryan, **20**
- 111-11-1111, Gabriel, **21**
- 373-32-2111, Joshua, **40**
- 222-22-2222, Homer, **50**

Sorted File
Restaurant DBA knows filtering on <u>age</u> dominates most searches. Sorts tuples on age

Sorted File Summary

- Pros
  - Good for filtering-type queries
    - Best for range queries (WHERE age > 21)
    - Pretty good for point (exact-match) queries (WHERE age = 21)
  - Scanning the file also doesn't slow-down compared to heap files
  - Deletion also faster: you normally need to search for a tuple or range of tuples to remove.
- Cons:
  - Bad for insertion-heavy workloads

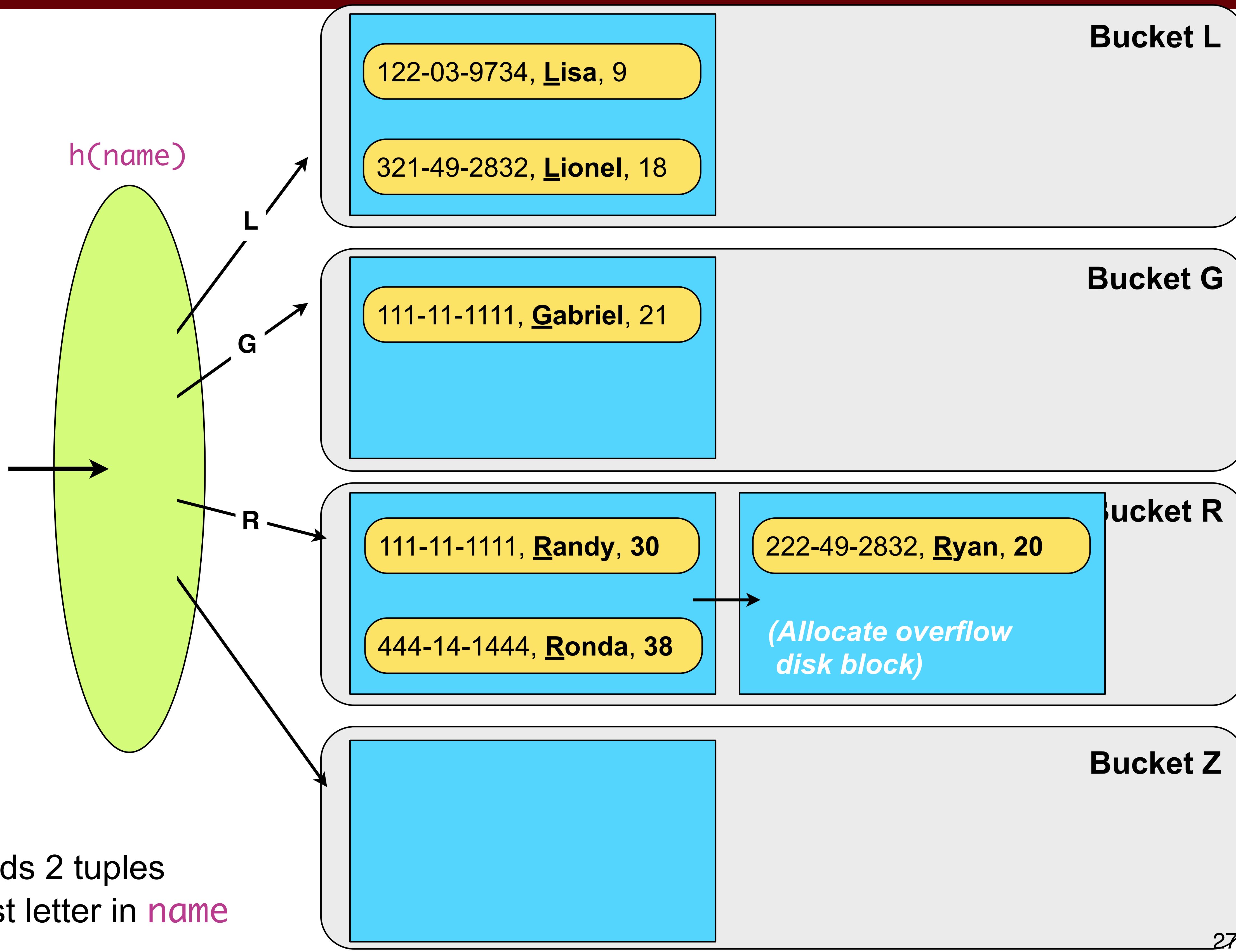
Option 3: Hashed Files



Assume:

- Each block holds 2 tuples
- `h(name)` = first letter in `name`

Suppose We Hash on the name Attribute



Assume:
- Each block holds 2 tuples
- h(name) = first letter in name

How to Minimize Overflow Blocks?

- Overflows should be avoided when possible
  - Long overflow chains represent a mini heap file!!
  - Search performance just got *a lot worse* in those buckets!

- **Goal:** Minimize Overflow Blocks (We want *shallow* buckets)
  - We want a hash function h(k) that maps search-keys to buckets with *uniform distribution*
    - Each bucket is assigned with approximately equal number of tuples
    - We want all buckets to be as full as possible
      - Minimize "skew" in some buckets
      - The `mod` operator typically produces uniform distribution

Static Hashing Example

- Consider this example:
  - Let the search key be age
  - Hash function `h(k) = k % 2`
    - Two buckets: 0 and 1
  - Assume: each 4KB block only holds 2 tuples (these are big tuples)

- Insert tuples with these *age* values:
  - 40, 10, 9, 16, 18, 20, 15, 30, 14, 1

- Again, but with `h(k) = k % 3`
  - *Could I have changed the hash function half-way through? (Nope! "Static" Hashing)*

Hashed File Summary

- Pros
  - Fast equality (or point) queries:
    - WHERE userID = '2'
  - Inserts and deletes are fast (hash the tuple's key, get bucket)
- Cons:
  - Back to linear search for range queries
  - Performance depends on a good hash function chosen right from the start

Topics

- Database File Structure
  - Fixed-Length Tuples
  - Variable-Length Tuples
- Tuple Organization in Files
  - Heap, Sorted, Hashed
- Cost Analysis
- Join Processing
  - Nested-Loop Join (NLJ)
  - Hash Join (HJ)
  - Sort-Merge Join (SMJ)

Let’s Analyze the Cost of Operations

- Assumptions
  - INSERT and DELETE only deal with a single tuple
  - Point (equality) queries are made on a primary key, so there's just 1 tuple matching the WHERE clause.
- More Assumptions
  - Heap Files:
    - Insert always go to the end of the file
  - Sorted Files:
    - Files are shifted before insertion and compacted after deletion
    - Select statements are always made on search-keys
  - Hashed Files:
    - Selects are always made on search-keys; search-keys are hashed

Cost of Common Operations (Worst Case)

- *B* = The number of blocks used to store the table's file on disk
- *D* = Average Data Access Time (DAT) to read/write a disk block

Operation	Heap File	Sorted File	Hashed File
Scan all tuples	*BD* ✓	*BD* ✓	*BD* ✓
Point Selection (PT) WHERE attr = 'val'	*BD*	*log(B) x D*	*(1 + overflow blocks in bucket h(k)) x D* ✓*
Range Selection WHERE attr < val	*BD*	*[log(B) + blocks with matches] x D* ✓	*BD*
Insert	*2D* ✓	*PT + BD*	*PT + D* ✓*
Delete	*PT + D* ✓	*PT + BD*	*PT + D* ✓*

`* Assumes good hash function ==> overflow buckets < log_2(B)`

Topics

- Database File Structure
  - Fixed-Length Tuples
  - Variable-Length Tuples
- Tuple Organization in Files
  - Heap, Sorted, Hashed
- Cost Analysis
- Join Processing
  - Nested-Loop Join (NLJ)
  - Hash Join (HJ)
  - Sort-Merge Join (SMJ)

Cost of Join Operators?

- Natural joins are one of the most common operations
  - But we alluded to the fact that tuple organization greatly affects JOINs
- There are three classic natural-join algorithms:
  - Nested-Loop Join
  - Hash Join
  - Sort-Merge Join

Assumptions

- A natural join is taken between relations R and S
- There is one common attribute between R and S, named **comm**
- The value of an attribute A in a tuple *t* is denoted $t.A$
- Performance is again measured in terms of block access:
  - B_R is the number of blocks used to store R on file
  - B_S is the number of blocks used to store S on file
  - D is the DAT to fetch a single block from disk

Nested Loop Join (NLJ)

- Simplest algorithm:
  - Loop through all tuples in **R** and in **S**
  - Check for equality on **R.comm** and **S.comm**
    - Concatenate tuples if true

```
public Relation nestedLoopJoin(Relation R, Relation S) {
   Relation T = new Relation();
   for each tuple r in R
      for each tuple s in S
         if (r.comm == s.comm)
            create new Tuple(r, s) and add it to T
   return T;
}
```

Evaluation of NLJ

- Pros:
  - Works with all file types (heap files, sorted files, hash files)
  - No need pre-process files before running

- Cons:
  - Really slow: $O(B_R \cdot B_S) \cdot D$

Topics

- Database File Structure
  - Fixed-Length Tuples
  - Variable-Length Tuples
- Tuple Organization in Files
  - Heap, Sorted, Hashed
- Cost Analysis
- Join Processing
  - Nested-Loop Join (NLJ)
  - Hash Join (HJ)
  - Sort-Merge Join (SMJ)

Hash Join (HJ)

```
public Relation hashJoin(Relation R, Relation S) {
   Relation T = new Relation();

   // Phase I: Build hash map
   HashMap map = new HashMap();
   for each tuple r in R {
      if (!map.containsKey(r.comm))
        list = new List();
      else
        list = map.get(r.comm);
     list.add(r);
     map.put(r.comm, list);
   }

   // Phase II: Join up with S
   for each tuple s in S {
      if (s.comm in map) {
         list = map.get(s.comm)
         for-each tuple r in list:
            create new Tuple (r, s) and add to T
      }
   }
   return T;
}
```

Evaluation of HJ

- Pros:
  - Fast! Just a single scan of both files $O(B_R + B_S) \cdot D$
  - Works with all file types (heap, sorted, hashed)
- Cons:
  - High space complexity
    - Need to store HashMap completely in memory
    - That's the entire file of R!

Topics

- Database File Structure
  - Fixed-Length Tuples
  - Variable-Length Tuples
- Tuple Organization in Files
  - Heap, Sorted, Hashed
- Cost Analysis
- Join Processing
  - Nested-Loop Join (NLJ)
  - Hash Join (HJ)
  - Sort-Merge Join (SMJ)

Sort-Merge Join (SMJ)

```
public Relation sortJoin(Relation R, Relation S) {
   Relation T = new Relation()
   if (R is not sorted on R.comm)
       sort R on R.comm
   if (S is not sorted on S.comm)
       sort S on S.comm
   int i = 0, j = 0;
   while (i < R.size() && j < S.size()) {
      // Match found, enter merge phase
      if (R[i].comm == S[j].comm) {
         while (R[i].comm == S[j].comm && i < R.size()) {
            k = j;
            while (R[i].comm == S[k].comm && k < S.size()) {
               create new Tuple (R[i], S[k]) and add it to relation T
               k++;
            }
            i++;
         }
      }
      else if (R[i].comm < S[j].comm)
         i++;
      else
         j++;
   }
   return T;
}
```

SMJ (Cont.)

R

```
--------------------
A | B | C | D | ...
```

i

C
c1
c1
c2
c2
c3
c3
c4
c4

Assume: $i < j \implies c_i < c_j$

S

```
--------------------
X | Y | Z | C | ...
```

j

C
c0
c1
c1
c2
c3
c4

SMJ (Cont.)

```
R
--------------------
A | B | C | D | ...
i         c1
          c1
          c2
          c2
          c3
          c3
          c4
          c4
```

Assume: $i < j \implies c_i < c_j$

```
S
--------------------
X | Y | Z | C | ...
j             c0
              c1
              c1
              c2
              c3
              c4
```

SMJ (Cont.)

R

A | B | C | D | ...

i

c1

c1

c2

c2

c3

c3

c4

c4

Assume: $i < j \implies c_i < c_j$

S

X | Y | Z | C | ...

c0

j

c1

c1

c2

c3

c4

k

SMJ (Cont.)

```
R
--------------------
A | B | C | D | ...
        c1      <- i
        c1
        c2
        c2
        c3
        c3
        c4
        c4
```

Assume: $i < j \implies c_i < c_j$

```
S
--------------------
X | Y | Z | C | ...
            c0
            c1      <- j
            c1      <- k
            c2
            c3
            c4
```

SMJ (Cont.)

R
```
--------------------
A | B | C | D | ...
```
i

c1
c1
c2
c2
c3
c3
c4
c4

Assume: $i < j \implies c_i < c_j$

S
```
--------------------
X | Y | Z | C | ...
```
j

k

c0
c1
c1
c2
c3
c4

SMJ (Cont.)

R

A | B | C | D | ...

c1
c1
c2
c2
c3
c3
c4
c4

i

j

S

X | Y | Z | C | ...

c0
c1
c1
c2
c3
c4

k

Assume: $i < j \implies c_i < c_j$

SMJ (Cont.)

```
R
--------------------
A | B | C | D | ...
        c1
        c1
        c2
        c2
        c3
        c3
        c4
        c4
```

i

Assume: $i < j \implies c_i < c_j$

```
S
--------------------
X | Y | Z | C | ...
            c0
            c1
            c1
            c2
            c3
            c4
```

j

k

SMJ (Cont.)

R

```
--------------------
A | B | C | D | ...
        c1
i       c1
        c2
        c2
        c3
        c3
        c4
        c4
```

Assume: $i < j \implies c_i < c_j$

S

```
--------------------
X | Y | Z | C | ...
j             c0
              c1
              c1
k             c2
              c3
              c4
```

SMJ (Cont.)

```
R
--------------------
A | B | C | D | ...
          c1
          c1
i         c2
          c2
          c3
          c3
          c4
          c4
```

Assume: $i < j \implies c_i < c_j$

```
S
--------------------
X | Y | Z | C | ...
              c0
j             c1
              c1
              c2
              c3
              c4
```

SMJ (Cont.)

```
R
---------------------
A | B | C | D | ...
          c1
          c1
i         c2
          c2
          c3
          c3
          c4
          c4
```

Assume: $i < j \implies c_i < c_j$

```
S
---------------------
X | Y | Z | C | ...
              c0
j             c1
              c1
              c2
              c3
              c4
```

SMJ (Cont.)

R

```
--------------------
A | B | C | D | ...
```

c1
c1
c2
c2
c3
c3
c4
c4

i

Assume: $i < j \implies c_i < c_j$

S

```
--------------------
X | Y | Z | C | ...
```

c0
c1
c1
c2
c3
c4

j

SMJ (Cont.)

```
R
--------------------
A | B | C | D | ...
        c1
        c1
        c2
        c2
        c3
        c3
        c4
        c4
```

i

```
S
--------------------
X | Y | Z | C | ...
            c0
            c1
            c1
            c2
            c3
            c4
```

j

k

Assume: $i < j \implies c_i < c_j$

SMJ (Cont.)

```
R
--------------------
A | B | C | D | ...
        c1
        c1
        c2
        c2
        c3
        c3
        c4
        c4
```

i

```
S
--------------------
X | Y | Z | C | ...
            c0
            c1
            c1
            c2
            c3
            c4
```

j

k

Assume: $i < j \implies c_i < c_j$

SMJ (Cont.)

```
R
--------------------
A | B | C | D | ...
        c1
        c1
        c2
        c2
        c3
        c3
        c4
        c4
```

... continues...

i

j

```
S
--------------------
X | Y | Z | C | ...
            c0
            c1
            c1
            c2
            c3
            c4
```

... continues...

k

Evaluation of SMJ

- Pros:
  - Performs well if files are both pre-sorted on common attribute!
    - Best case: $O(B_R + B_S) \cdot D$, if there are few matches
      - Same as HJ without the auxiliary space complexity!

- Cons:
  - What if, for every common attribute value, there's a match?
    - Back to $O(B_R \cdot B_S) \cdot D$ *(Why? SMJ basically NLJ)*
  - Should only use this if tuples are sorted on common attributes!
    - Would need to sort both relations first! $O(B_R \cdot log_2(B_R) + B_S \cdot log_2(B_S)) \cdot D$
    - Worse space complexity than HJ too: Need to bring in both relations for sorting

Topics

- Tuple Organization in Files
  - Heap, Sorted, Hashed
- Cost Analysis
- Join Processing
  - Nested-Loop Join (NLJ)
  - Hash Join (HJ)
  - Sort-Merge Join (SMJ)
- Conclusion

In Conclusion...

- Spatial and temporal locality of algorithms enable caching
  - Leads to storage hierarchy
  - Leads to fast, affordable computer systems with lots of storage

- Hard disk drives are non-volatile and cheap
  - Performance and reliability characteristics

In Conclusion... (Cont.)

- We talked about ways to structure our tuples
  - Fixed and variable tuples
- The way we organize tuples within files matters to performance!
  - Heap, sorted, hashed
  - Pros & cons of each? (When would you consider using each?)

Administrivia 11/5

- Reminders
  - Hwk 5 due tonight
    - Q #5 confusions -- you're not confused. I gave away too much info in the problem.
  - Exam 2 next Friday
- Last time...
  - Started file manager
  - Tuple formatting: fixed vs. variable-length support
- Today:
  - Tuple organization within files

Administrivia 11/6

- Reminders
  - Hwk 6 posted
    - Due 11/18
  - Project 4 posted
    - Due 12/6
  - Exam 2 next Friday
- Last time...
  - Finished disk optimizations: disk scheduling, buffer mgr
  - Started file manager
  - Today: file structure

Administrivia 11/8

- Reminders
  - Hwk 6 posted
    - Due 11/18 (next Friday)
  - Project 4 posted
    - Due 12/6 (Monday of reading week)
  - Exam 2 on Friday
- Last time...
  - Tuple organization in files
- Today...
  - Join algorithms