Storage

Holger Pirk

Slides as of 10/02/22 09:24

We are going down the rabbit hole now

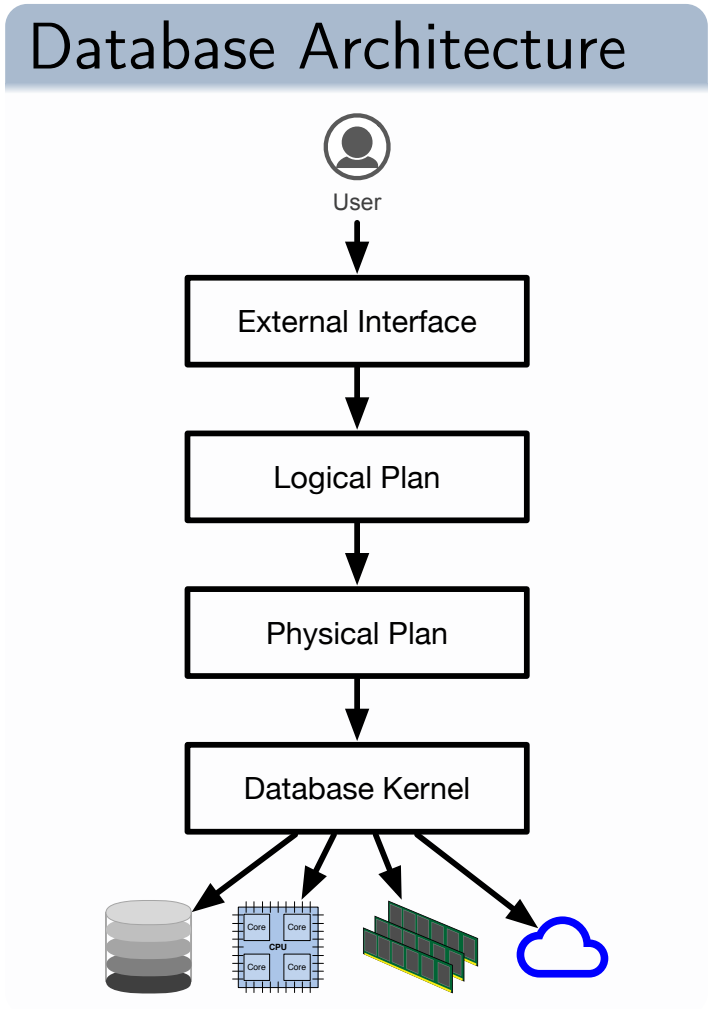
- We are crossing the threshold into the system
- (No more rules without exceptions :-)

Purpose of this lecture

Understand storage alternatives

- In-memory vs. disk
- N-ary vs. decomposed storage
- Main/Delta storage
- Catalog management
- Buffer management

DBMS architecture

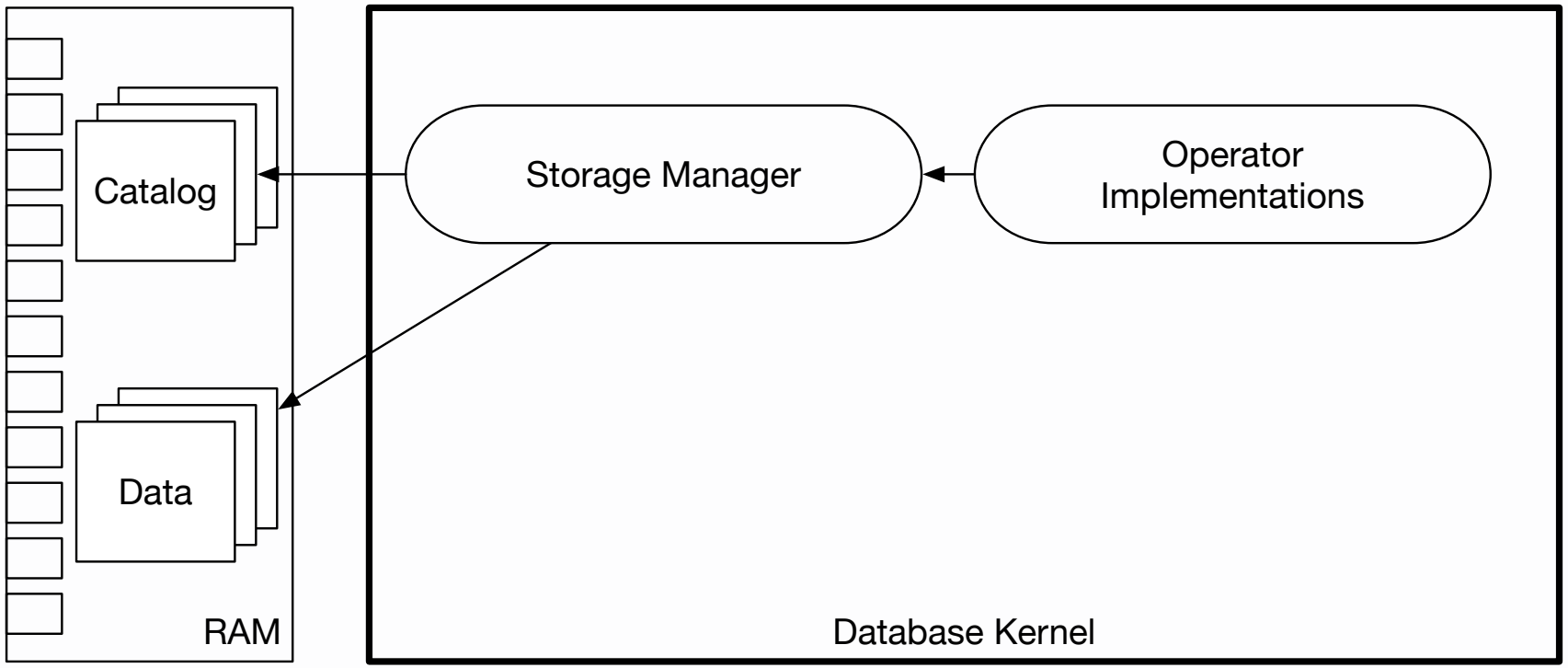


So, what is a database kernel

- The key part of a data management system – everything else is optional
- A library of core functionality: I/O, memory management, operators, etc.
  - sound familiar?
  - (Yes, it is quite similar to an OS kernel)
- Provides an interface to subsystems

Let's open up the kernel (top-down)

A (preliminary) database kernel architecture



Let's open up the kernel

The kernel interface

```
class StorageManager;
class OperatorImplementations;
class DatabaseKernel{
        StorageManager& getStorageManager();
        OperatorImplementations& getOperatorImplementations();
}
```

Let's start with storage

- The first operation every database needs to support is inserting new tuples
- Inserts are usually not optimized (let's ignore concurrency control for now)
- Arrive at the storage layer as intact tuples

Inserting Data

Schema

```
create table Employee (id int, name varchar, salary int, joiningDate int);
```

In SQL

```
INSERT INTO "Employee" VALUES(4,'holger',100000,43429342);
SELECT * FROM "Employee" WHERE id=4;
DELETE FROM "Employee" WHERE name='holger';
```

In C++ (as we would like it)

```
StorageManager().getTable("Employee").insert({4, "holger", 100000, 43429342});
StorageManager().getTable("Employee").findTuplesWithAttributeValue(id, 4);
StorageManager().getTable("Employee").deleteTuplesWithAttributeValue(name, "holger");
```

Storage Manager Interface

In C++

```
struct Table;
class StorageManager {
  map<string, Table> catalog;

public:
  Table& getTable(string name) { return catalog[name]; };
};
```

Storing tuples

Many decisions need to be made

- **Where** to store data
  - **Disk**
  - **Memory**
  - (Tape)
  - (Non-volatile memory)
  - (etc.)
- **How** to store data
  - N-ary or decomposed?
  - Sorted?
  - Hybrid?
  - Dictionary-compressed?
  - RLE-encoded?
  - (and many more)

The interface

Simplified Table Interface

```
struct Table {
  using AttributeValue = variant<int, float, string>;
  using Tuple = vector<AttributeValue>;

  void insert(Tuple) {};
  vector<Tuple> findTuplesWithAttributeValue(int attributePosition, AttributeValue) {};
  void deleteTuplesWithAttributeValue(int attributePosition, AttributeValue) {};
};
```

How to store tuples

Fundamental Mismatch

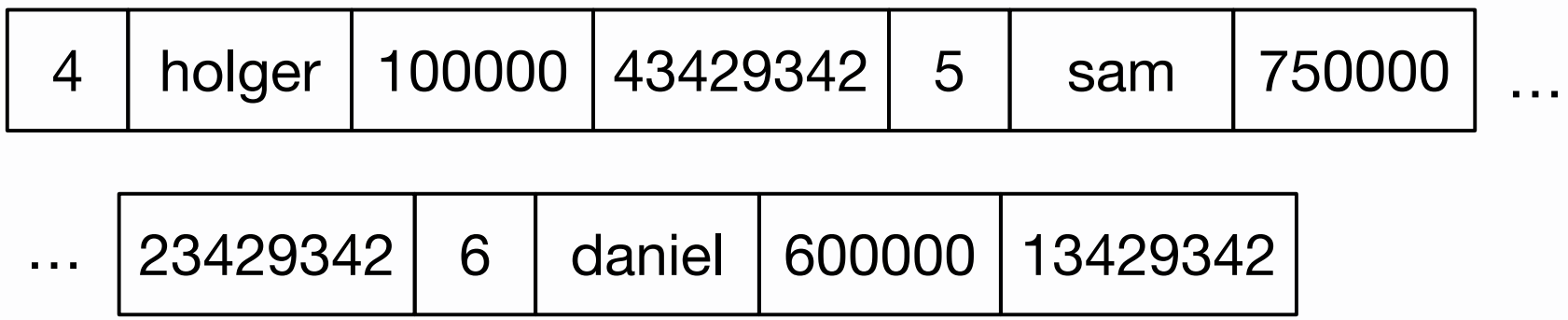
- Relations are two-dimensional
- Memory is one-dimensional
- Tuples need to be linearized
- There are two mainstream strategies
  - (and research on hybrids)

The N-ary Storage Model (NSM)

Example

```
StorageManager().getTable("employees").insert({4, "holger", 100000, 43429342});
StorageManager().getTable("employees").insert({5, "sam", 750000, 23429342});
StorageManager().getTable("employees").insert({6, "daniel", 600000, 13429342});
```

Linearization in N-ary format



- In marketing they call this a row store

N-ary storage implementation

NSM Table

```
struct NSMTable : public Table {
  struct InternalTuple {
    Tuple actualTuple;
    bool deleted = false; // this will be handy later
    InternalTuple(Tuple t) : actualTuple(t) {}
  };
  vector<InternalTuple> data;
  //
  void insert(Tuple);
  vector<Tuple> findTuplesWithAttributeValue(int attributePosition, AttributeValue);
  void deleteTuplesWithAttributeValue(int attributePosition, AttributeValue);
};
```

NSM Table insert

```
void NSMTable::insert(Tuple t) { data.push_back(t); }
```

N-ary storage manager implementation

NSM Table find (first shot)

```
vector<Table::Tuple> NSMTable::findTuplesWithAttributeValue(int attributePosition,
                                                            AttributeValue value) {
  vector<Tuple> result;
  for(size_t i = 0; i < data.size(); i++) {
    if(data[i].actualTuple[attributePosition] == value)
      result.push_back(data[i].actualTuple);
  }
  return result;
};
```

Handling deletes

NSM Table delete

```
void NSMTable::deleteTuplesWithAttributeValue(int attributePosition, AttributeValue value) {
  for(size_t i = 0; i < data.size(); i++)
    if(data[i].actualTuple[attributePosition] == value)
      data[i].deleted = true;
}
```

Handling deletes

NSM Table find for real

```
vector<Table::Tuple> NSMTable::findTuplesWithAttributeValue(int attributePosition,
                                                            AttributeValue value) {
  vector<Tuple> result;
  for(size_t i = 0; i < data.size(); i++) {
    if(data[i].actualTuple[attributePosition] == value && !data[i].deleted)
      result.push_back(data[i].actualTuple);
  }
  return result;
};
```

Is that a good idea?

It depends!

Impact of data locality

Locality is king

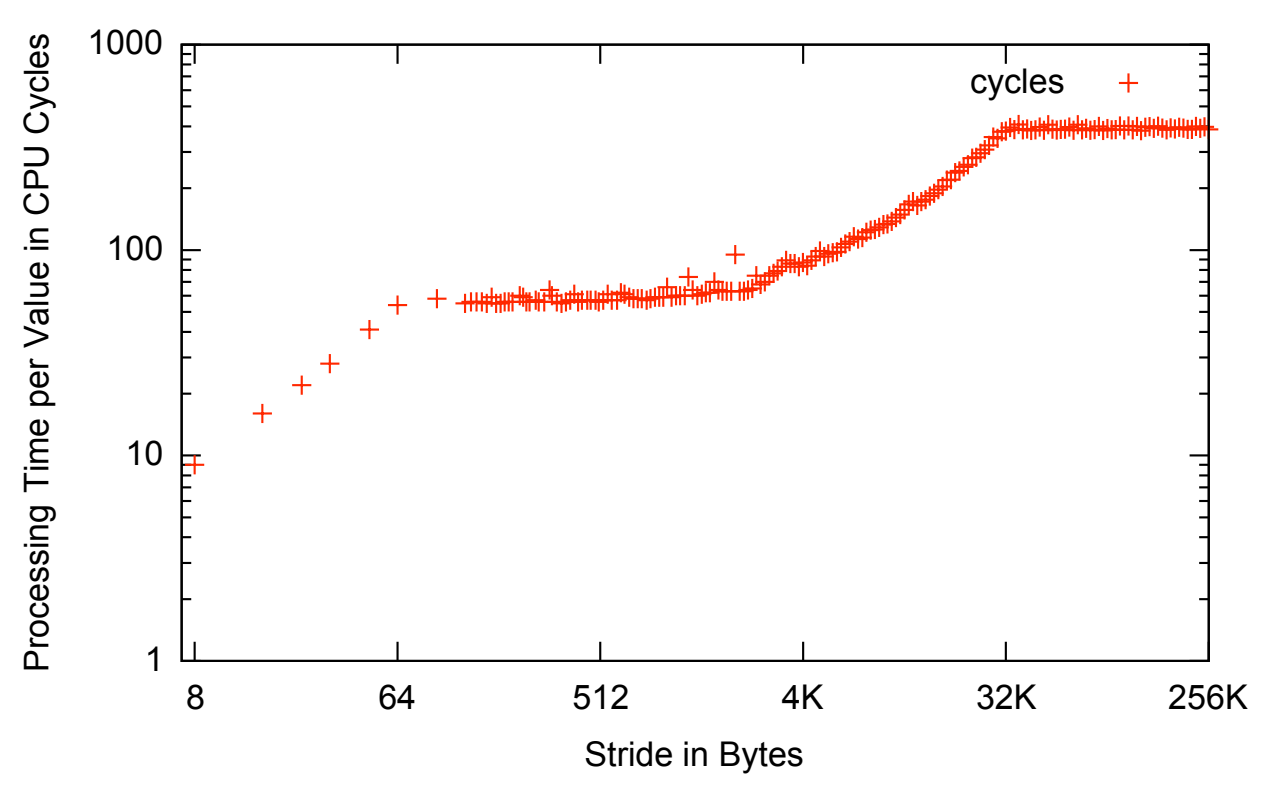
- But why? Aren't we talking about in-memory databases?
- Doesn't RAM have constant access latency?

Impact of data locality

A simple experiment

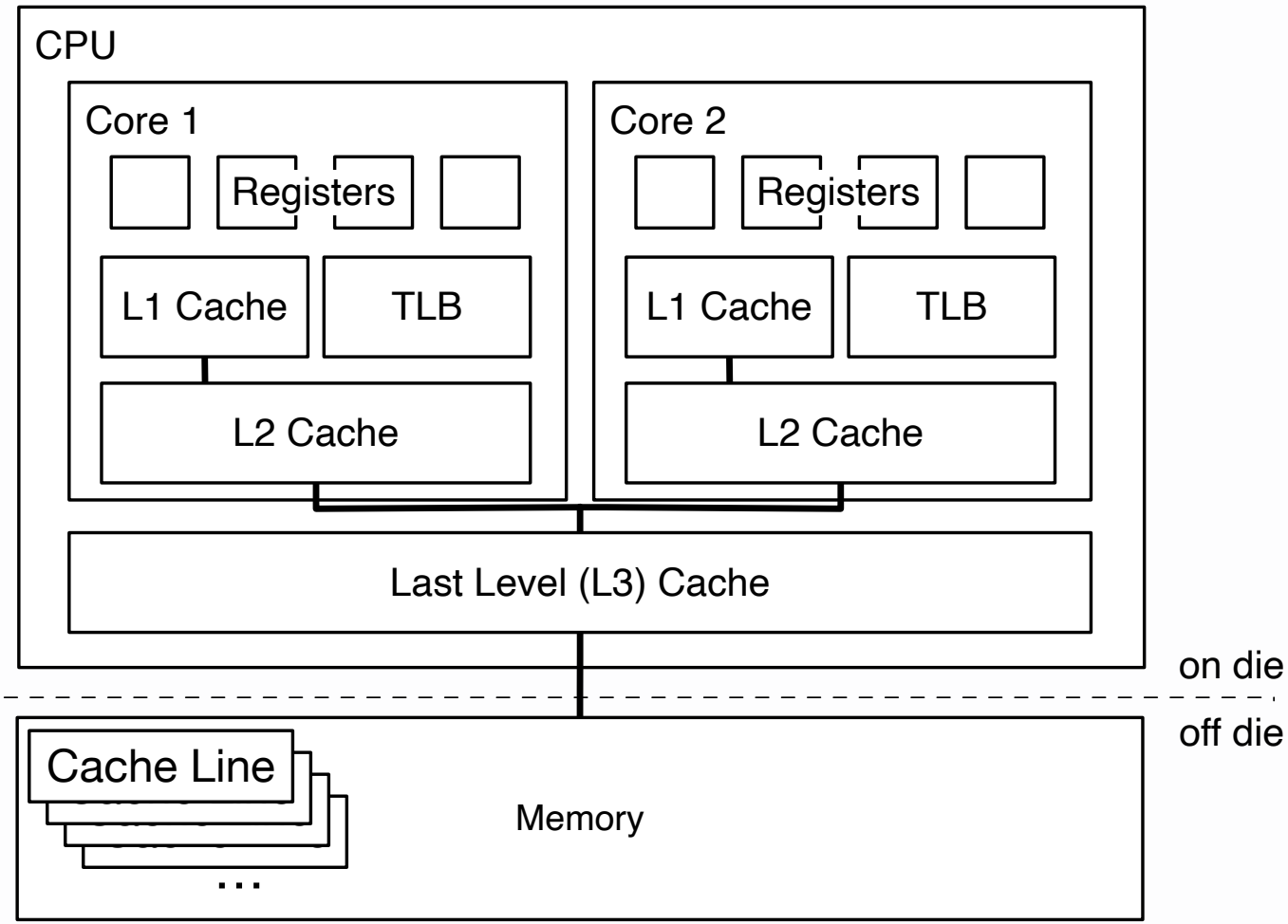
```
for(size_t i = 0; i < storage.size(); i++)
  if(storage[i][0] == value) // happens to be always false in this experiment
    result.push_back(storage[i]);
```

Locality to access time



Why data locality matters

A CPU from the inside



Why data locality matters

Nomenclature

- We call units of data transfer blocks, pages or cache lines

When N-ary storage works well

Linearization in N-ary format

4	holger	100000	43429342	5	sam	750000

...

... | 23429342 | 6 | daniel | 600000 | 13429342 |

Insert a new tuple

```
employees.insert({3,"peter",200000,33429342});
```

Simply append the tuple

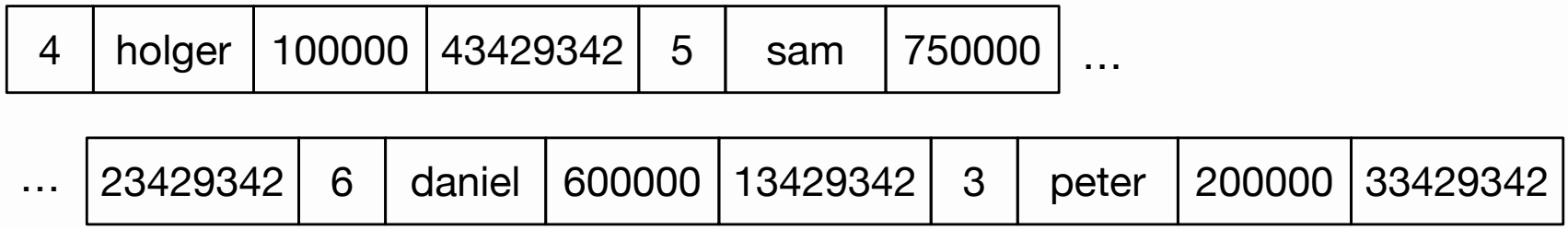
4	holger	100000	43429342	5	sam	750000

...

... | 23429342 | 6 | daniel | 600000 | 13429342 | 3 | peter | 200000 | 33429342 |

When N-ary storage works well

Linearization in N-ary format



Lookup a tuple by index

```
employees.data[2].actualTuple
```

When N-ary storage is suboptimal

The database

4	holger	100000	43429342	5	sam	750000	...

...	23429342	6	daniel	600000	13429342	3	peter	200000	33429342

They query

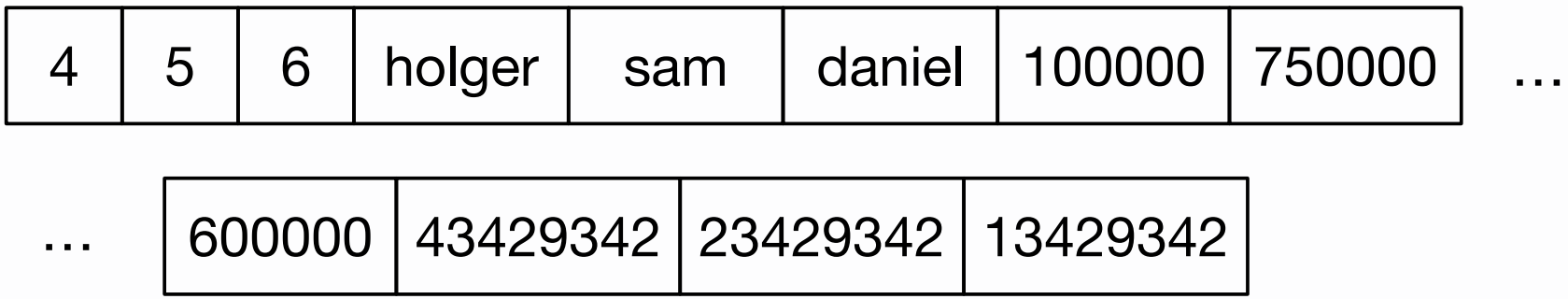
- find all tuples that have an salary of 50.000
- In SQL `select * from employee where salary = 50000;`

The Decomposed Storage Model (DSM)

Example

```
StorageManager().getTable("employees").insert({4, "holger", 100000, 43429342});
StorageManager().getTable("employees").insert({5, "sam", 750000, 23429342});
StorageManager().getTable("employees").insert({6, "daniel", 600000, 13429342});
```

Linearization in Decomposed format



- In marketing they call this a column store

DSM storage manager implementation

DSM Table

```
struct DSMTable : public Table {
  using Column = vector<AttributeValue>;
  vector<Column> data;
  vector<bool> deleteMarkers;
  //
  void insert(Tuple);
  vector<Tuple> findTuplesWithAttributeValue(int attributePosition, AttributeValue);
  void deleteTuplesWithAttributeValue(int attributePosition, AttributeValue);
};
```

DSM Table inserts cause tuple decomposition

```
void DSMTable::insert(Tuple tuple) {
  for(int i = 0; i < tuple.size(); i++)
    data[i].push_back(tuple[i]);
}
```

Handling deletes

DSM Table deletes

```
void DSMTable::deleteTuplesWithAttributeValue(int attributePosition, AttributeValue value) {
  for(size_t i = 0; i < data[attributePosition].size(); i++)
    if(data[attributePosition][i] == value)
      deleteMarkers[i] = true;
}
```

DSM storage manager implementation

DSM find requires tuple reconstruction

```
vector<Table::Tuple> DSMTable::findTuplesWithAttributeValue(int attributePosition,
                                                            AttributeValue value) {
  vector<Tuple> result;
  for(size_t i = 0; i < data[attributePosition].size(); i++)
    if(data[attributePosition][i] == value && !deleteMarkers[i]) {
      Tuple reconstructedTuple;
      for(int column = 0; column < data.size(); column++)
        reconstructedTuple.push_back(data[column][i]);
    }
  return result;
};
```

When decomposed storage works well

The database

4	5	6	holger	sam	daniel	100000	750000	...

...	600000	43429342	23429342	13429342

They query

- find all tuples that have an salary of 50.000
- In SQL `select * from employee where salary = 50000;`

When decomposed storage is suboptimal

Linearization in Decomposed format

4	5	6	holger	sam	daniel	100000	750000	...

...	600000	43429342	23429342	13429342

Insert a new tuple

```
employees.insert({3,"peter",200000,33429342});
```

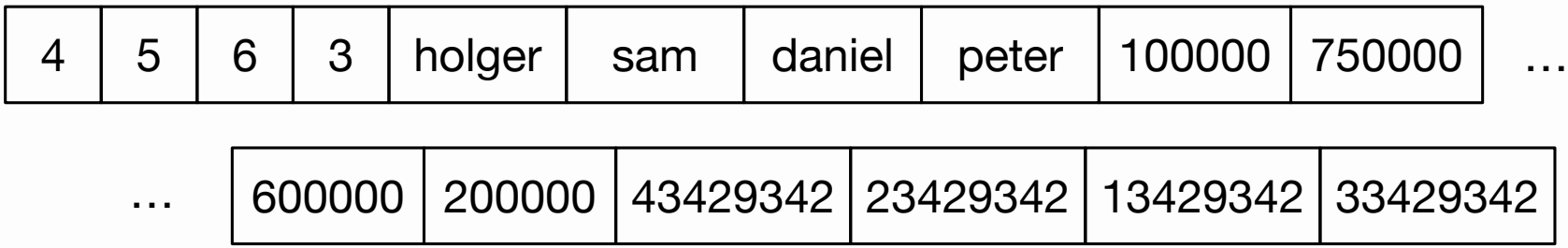
Tuples need to be decomposed before insertion

4	5	6	3	holger	sam	daniel	peter	100000	750000	...

...	600000	200000	43429342	23429342	13429342	33429342

When decomposed storage is suboptimal

Tuples are decomposed in database



Access a single tuple

```
select * from employee where id = 6; -- assume index access
```

- tuple needs to be reconstructed:

```
Tuple reconstructedTuple;
for(int column = 0; column < data.size(); column++)
  reconstructedTuple.push_back(data[column][2]);
```

The bottom line on NSM vs. DSM

- DSM works well for scan-heavy queries
  - Those are very common in analytical processing (aggregate, join, . . .)
  - Analytics mostly operate on historical data
- NSM works well for lookups and inserts
  - Those are very common in transactional processing (insert sales item, lookup product)
  - Transactions mostly operate on recent data
- This naturally lead to the development of hybrids

Delta/Main storage manager implementation

Delta/Main Table

```
struct HybridTable : public Table {
  DSMTable main;
  NSMTable delta;
  void insert(Tuple t);
  vector<Tuple> findTuplesWithAttributeValue(int attributePosition, AttributeValue);
  void deleteTuplesWithAttributeValue(int attributePosition, AttributeValue);
  void merge();
};
```

Delta/Main Table insert

```
void HybridTable::insert(Tuple t) { delta.insert(t); };
```

Delta/Main storage manager implementation

Delta/Main Table lookup

```
vector<Table::Tuple> HybridTable::findTuplesWithAttributeValue(int attribute,
                                                               AttributeValue value) {
  vector<Table::Tuple> results = main.findTuplesWithAttributeValue(attribute, value);
  vector<Table::Tuple> fromDelta = delta.findTuplesWithAttributeValue(attribute, value);
  results.insert(results.end(), fromDelta.begin(), fromDelta.end());
  return results;
};
```

Delta/Main Table delete

```
void HybridTable::deleteTuplesWithAttributeValue(int attribute, AttributeValue value){
  main.deleteTuplesWithAttributeValue(attribute, value);
  delta.deleteTuplesWithAttributeValue(attribute, value);
};
```

Delta/Main storage manager implementation

In C++

```
void HybridTable::merge(){
  for (auto i = 0u; i < delta.data.size(); i++) {
    main.insert(delta.data[i].actualTuple);
    delta.data[i].deleted = true;
  }
};
```

The bottom line on Delta + Main

- Delta + Main aims to exploit the nature of database workloads:
  - analytics on historical data in DSM
  - transactions on recent data in NSM
- However, it needs regular migrations which might need to lock the database
- It also complicates lookups

Side note

Delta and Main used to be implemented in separate databases (with optimized DBMSs): a *transactional system* and a *data warehouse*

Example

The program

```
employees.insert({3, "peter", 200000, 33429342});
employees.insert({4, "holger", 100000, 43429342});
employees.insert({5, "sam", 750000, 23429342});
employees.insert({6, "daniel", 600000, 13429342});
```

The database

3	peter	200000	33429342	4	holger	100000	43429342	...

...	5	sam	750000	23429342	6	daniel	600000	13429342

An idea

- Real-life data follows patterns
- If you can recognize and exploit these patterns, you can be much more efficient
- But, you need to store and maintain such "metadata"
- Lots of metadata can be kept: type, min/max values, histograms, auto-correlation, ...
- Let's study two simple ones: sortedness and denseness

Maintaining metadata

In C++

```
class Table {
  vector<Tuple> storage;
  bool firstColumnIsSorted = true; // e.g., 1, 3, 4, 8, 10
  bool firstColumnIsDense = true;  // e.g., 3, 4, 5, 6, 7 -- stricter than isSorted

public:
  void insert(Tuple t) {
    id(storage.size() > 0) {
      firstColumnIsSorted &= (t[0] >= storage.back()[0]);
      firstColumnIsDense &= (t[0] == storage.back()[0] + 1);
    }
    storage.push_back(t);
  };
};
```

Exploiting Metadata

In C++

```
class Table {
  vector<Tuple> storage;
  bool firstColumnIsSorted = true;
  bool firstColumnIsDense = true; // stricter than isSorted

public:
  vector<Tuple> findTuplesWithAttributeValue( //
      int attribute, AttributeValue value) {
    if(attribute == 0 && firstColumnIsDense)
      return {data[value - storage.front()[0]]};
    else if(attribute == 0 && firstColumnIsSorted &&
            binary_search(data, attribute, value)[0] == value)
      return {binary_search(data, attribute, value)};
    vector<Tuple> result;
    for(size_t i = 0; i < data.size(); i++) {
      if(data[i].actualTuple[attribute] == value)
        result.push_back(data[i]);
    }
    return result;
  };
};
```

Example

3	peter
4	holger
5	sam
6	daniel
7	...
8	...
9	...

Re-Establishing metadata

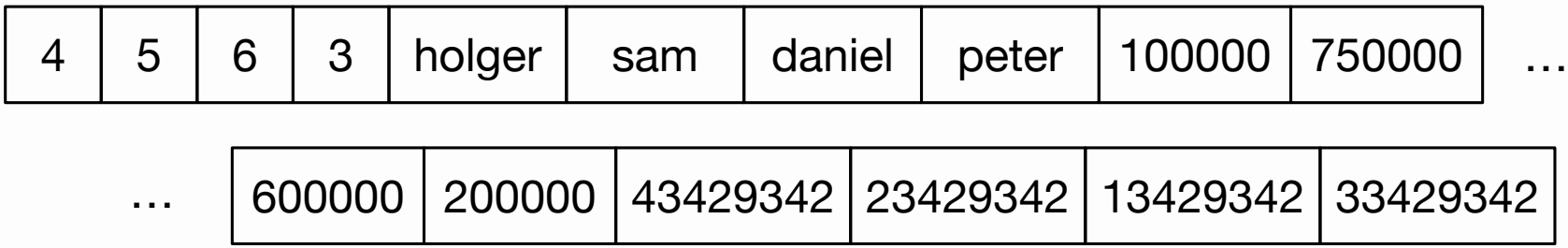
In C++

```
class Table {
  vector<Tuple> storage;
  bool isSorted = true;
  bool isDense = true;

public:
  void analyze() {
    sort(storage.begin(), storage.end(), [](auto l, auto r) { return l.id < r.id; });
    isDense = true;
    for(size_t i = 1; i < storage.size(); i++)
      isDense &= storage[i].id == storage[i - 1].id + 1;
  };
};
```

Let's think back to our example

Linearization in Decomposed format



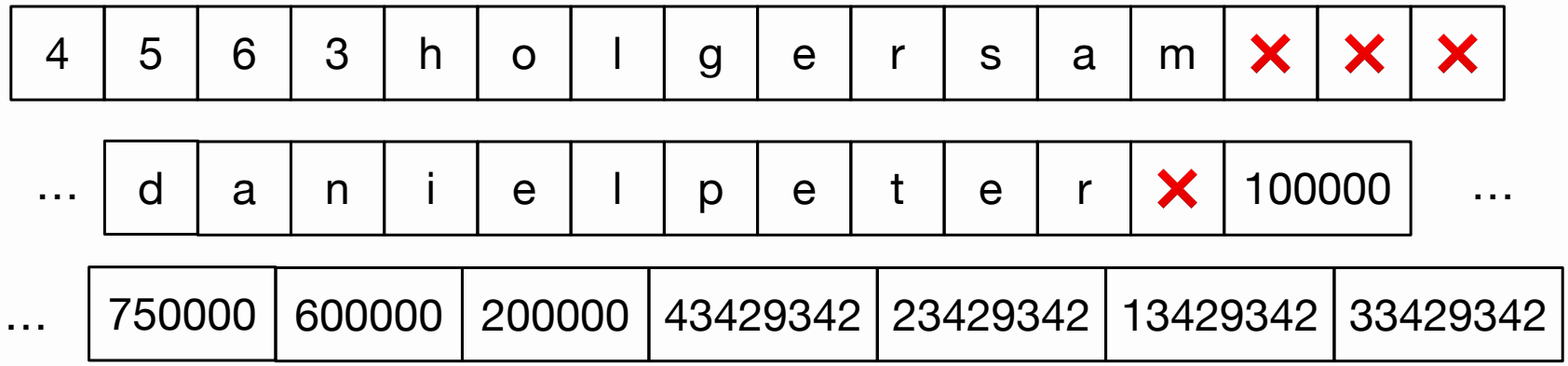
Names have different lengths

- $\rightarrow$ they occupy variable space in memory
- We would like to maintain fixed tuple sizes,
  - allows randomly access to tuples by their position
- Two choices
  - overallocate space for varchars (**many systems need a size parameter: e.g.,** `varchar(6)`)
  - store them out of place

Let us discuss these

In place storage

Our example database stored in in place DSM (name length <= 6)

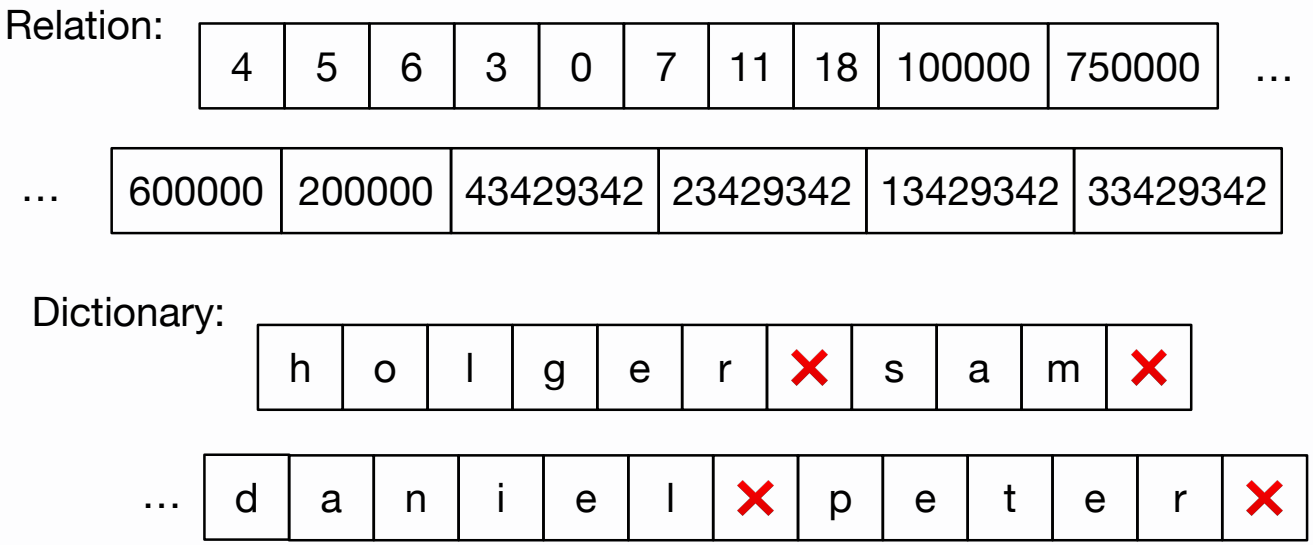


Properties

- Good for locality
- Simple
- Wastes space
  - particularly bad for in-memory databases

Out of place storage

Our example database stored in out of place DSM



Out of place storage

Properties

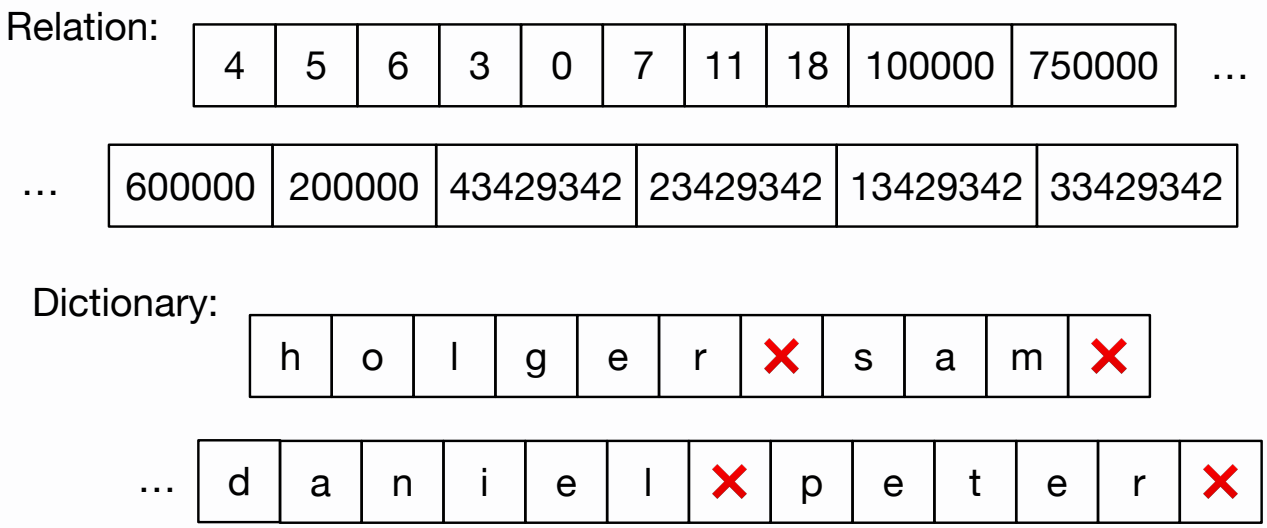
- Space conservative
- Bad for locality
- Complicated
  - Harder to implement
  - Garbage-collection is tricky
- Note: this is what most programming languages do

Nifty out of place storage: dictionary compression

Recall our database

4	holger	100000	43429342
5	sam	750000	23429342
6	daniel	600000	13429342
3	peter	200000	33429342

Our example database stored in out of place DSM



Nifty out of place storage: dictionary compression

A new tuple

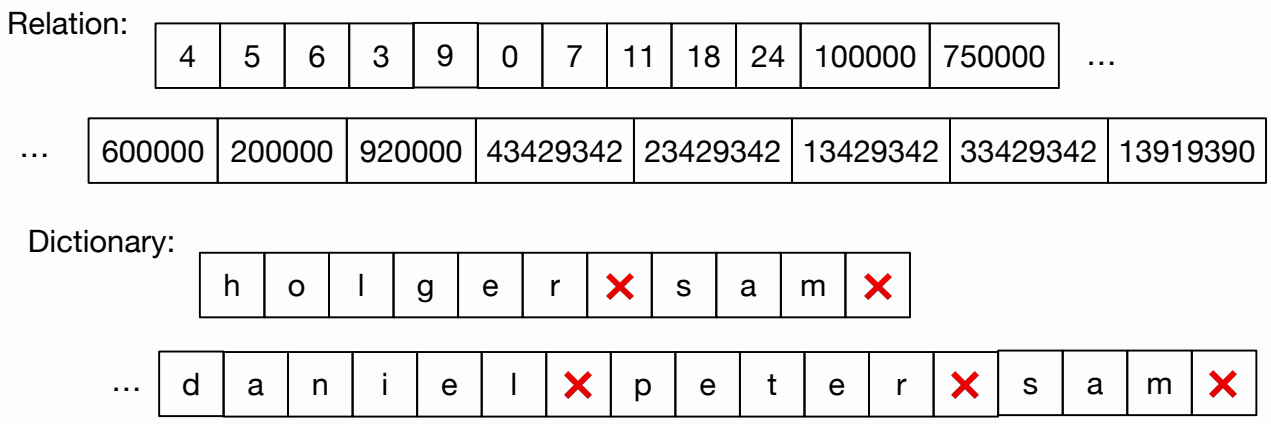
```
insert into employees values (9, "sam", 92000, 13919390);
```

Dictionary compression

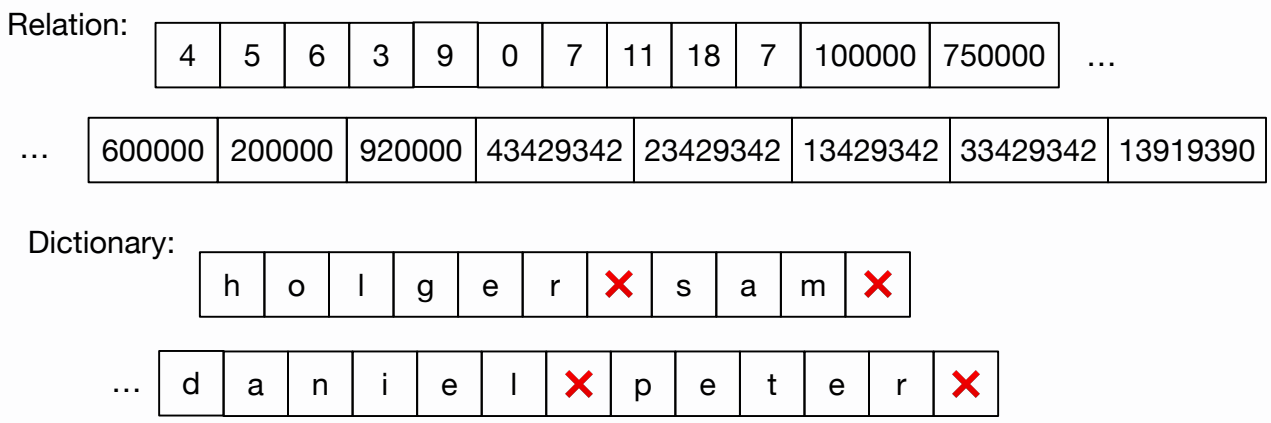
- Before every insert to the dictionary, check if the value is already present
  - If so, elide the insert and use the address of the existing value
  - Otherwise, insert the value

Nifty out of place storage: dictionary compression

Out of place DSM



Dictionary-compressed out of place DSM



What changes

How are disks different?

- Larger pages (Kilobytes instead of bytes)
- Much higher latency (ms instead of nanoseconds)
- Much lower throughput (hundreds of megabytes instead of tens of gigabytes per second)
  - This is why you think of DBMSs as I/O bound
- The operating system gets in the way
  - Filesize is limited $\rightarrow$ DBMS needs to map `tuple_ids` to files and offsets

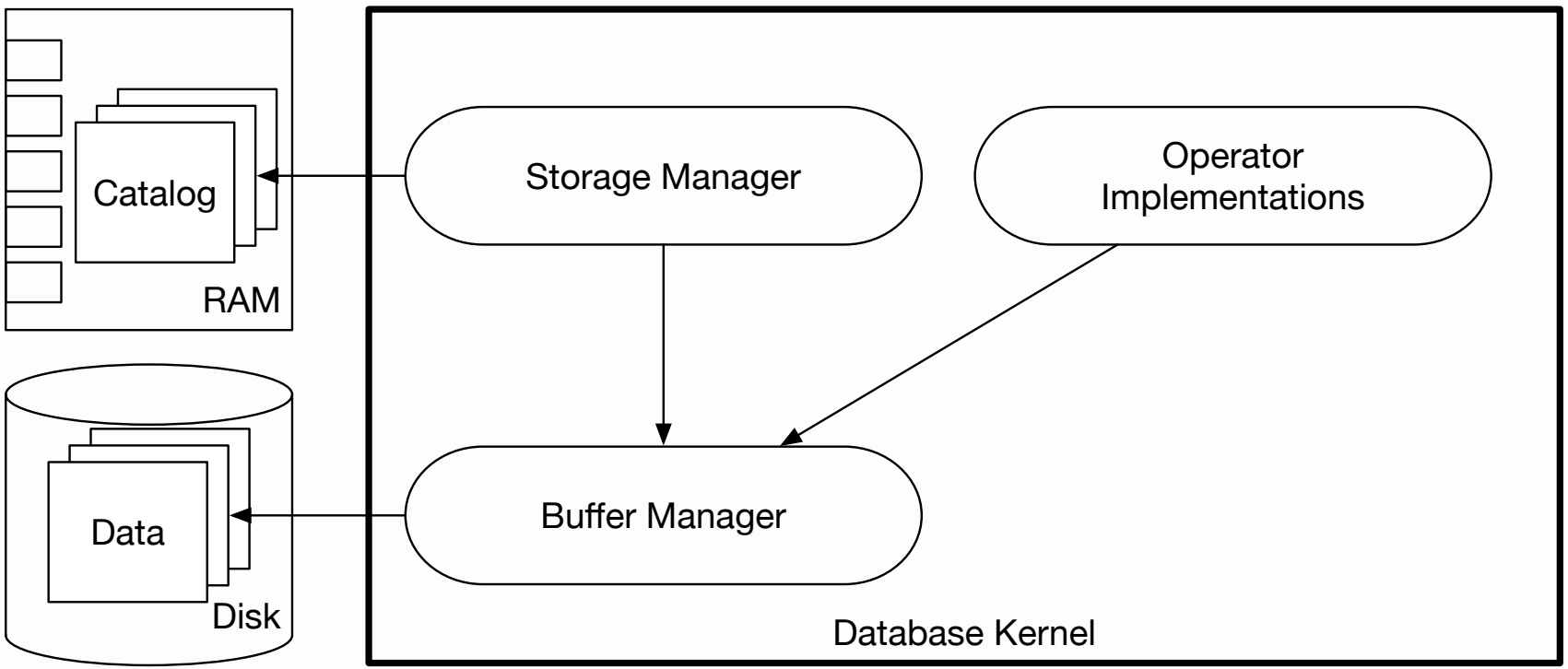
What changes

Goals shift

- Disks are dominating costs by far
  - Complicated I/O management strategies pay off
- Pages are large
  - Each page behaves like a mini-database
  - (in the case of N-ary storage)

Let's look back at the kernel

A (preliminary) database kernel architecture



What does a buffer manager do?

- Manages disk-resident data
  - Maps (unstructured) files to (structured) tables for reading and writing
  - Makes sure files have a fixed size (DBs don't trust OSs)
  - (Safely) writes data to disk when necessary
- Buffers writes in "open pages"
  - (open for writing, that is)
  - We make the simplifying assumption of one open page per relation (and that it can overflow slightly)

Disk-based storage manager implementation

In C++

```
class BufferManager;
class Table {
  BufferManager& bufferManager;
  string relationName = "Employee";

public:
  void insert(Tuple t) {
    bufferManager.getOpenPageForRelation(relationName).push_back(t); // insert tupe
    bufferManager.commitOpenPageForRelation(relationName);           // write to disk if
                                                                     // necessary
  };
  vector<Tuple> findTuplesWithAttributeValue(int attribute, AttributeValue value) {
    vector<Tuple> result;
    auto pages = bufferManager.getPagesForRelation(relationName);
    for(size_t i = 0; i < pages.size(); i++) {
      auto page = pages[i];
      for(size_t i = 0; i < page.size(); i++)
        if(page[i][attribute] == value)
          result.push_back(page[i]);
    }
    return result;
  };
};
```

The Buffer Manager

Interface

```
struct BufferManager {
  using Tuple = vector<AttributeValue>;
  using Page = vector<Tuple>;

  size_t tupleSize;
  map<string, vector<Tuple>> openPages; // maps to _exactly_ one open page (for simplicity)
  map<string, vector<string>> pagesOnDisk; // maps one relation to many pages on disk
  size_t numberOfTuplesPerPage();
  //
  vector<Tuple>& getOpenPageForRelation(string relationName);
  void commitOpenPageForRelation(string relationName);
  vector<Page> getPagesForRelation(string relationName);
};
```

The Buffer Manager

Implementation

```
vector<Tuple>& BufferManager::getOpenPageForRelation(string relationName) {
  return openPages[relationName]; // creates one if necessary
};

void BufferManager::commitOpenPageForRelation(string relationName) {
  while(openPages[relationName].size() >= numberOfTuplesPerPage(tupleSize)) {
    vector<Tuple> newPage;
    while(openPages[relationName].size() >
          numberOfTuplesPerPage) { // move overflowing tuples to new page
      newPage.push_back(openPages[relationName].back());
      openPages[relationName].pop_back();
    }
    pagesOnDisk[relationName].push_back(       //
        writeToDisk(openPages[relationName])); // write page
    openPages[relationName] = newPage;         // keep remaining tuples
  }
};
```

The Buffer Manager

Implementation

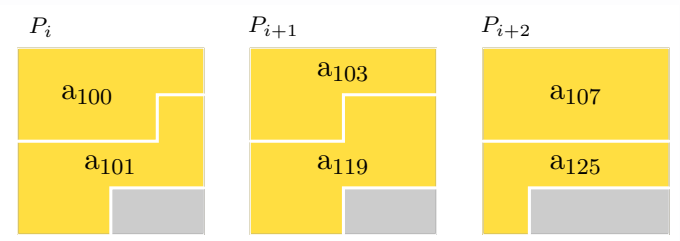
```
vector<vector<Tuple>> BufferManager::getPagesForRelation(string relationName) {
  vector<vector<Tuple>> result = {openPages[relationName]};
  for(size_t i = 0; i < pagesOnDisk[relationName].size(); i++)
    result.push_back(readFromDisk(pagesOnDisk[relationName][i]));
  return result;
};
```

Pages like mini-databases: Unspanned Pages

Goals

- Simplicity
- Random access performance (assuming known page fill-factors)
  - Given a `tuple_id`, find the record with a single page lookup

Example



Disadvantage

- Space waste
  - Cannot deal with large records
  - No in-page random access for variable-sized records

Unspanned Pages: Implementation & Calculation

Implementation

```
const long pageSizeInBytes = 4096;
size_t BufferManager::numberOfTuplesPerPage() { //
  return floor(pageSizeInBytes / tupleSizeInBytes);
}
```

If I ask for the space consumption of a relation of n tuples

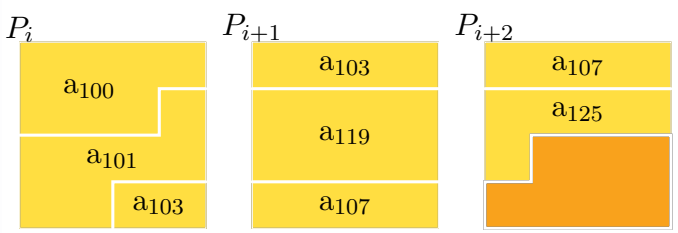
```
ceil(data.size() / numberOfTuplesPerPage())
```

Spanned Pages: optimizing for space efficiency

Goals

- Minimize space waste
- Support large records

Example



Disadvantages

- Complicated
- Random access performance
- No in-page random access for variable-sized records

Spanned Pages: Implementation

Calculating the number of pages per relation

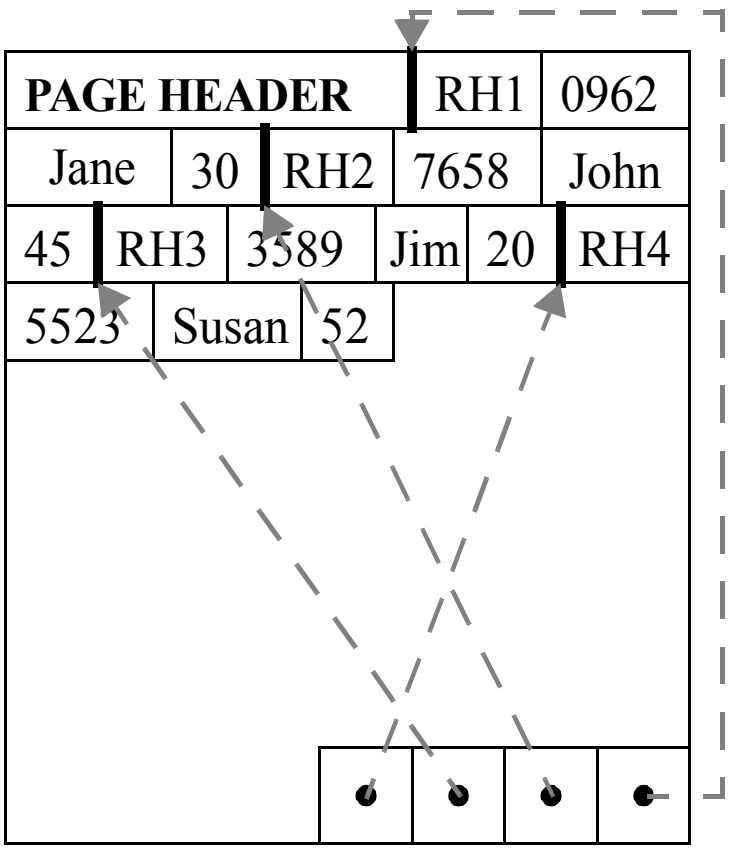
```
long dataSizeInBytes = tupleSizeInButes * data.size();
long numberOfPagesForTable = ceil(dataSizeInBytes / pageSizeInBytes);
```

Calculating the number of tuples per page

- Not a constant so not possible

Slotted Pages: Random Access for In-Place NSM

Enabling In-page random access



Explained

- Store tuples in in-place N-ary format (spanned or unspanned)
- Store tuple count in page header
- Store offsets to every tuple
  - Offsets only need to be typed large enough to address page
    - Bytes for pages smaller than 256 Bytes
    - Shorts for pages smaller than 65,536 Bytes
    - Integers for pages smaller than 4 Gigabytes

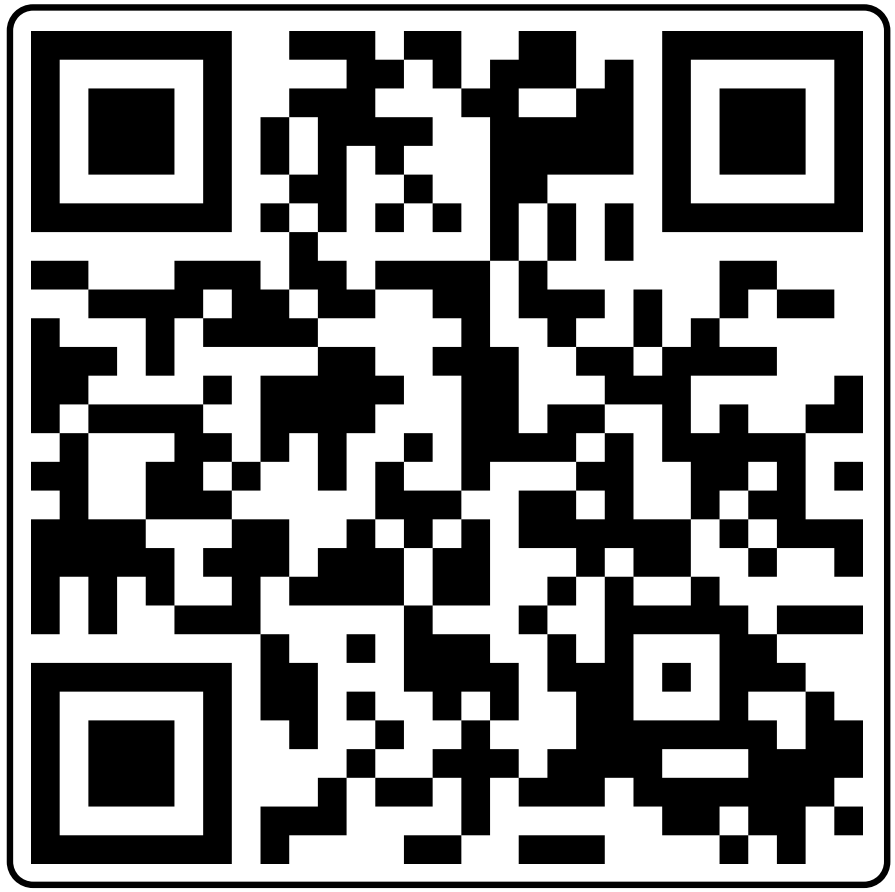
In-page dictionaries

Some disk-based database systems keep a dictionary per page

- This solves the problem of variable sized records
- Also allows duplicate elimination
- Question: Why don't they keep a global dictionary

Thank You!

Provide feedback, please!



https://co572.pages.doc.ic.ac.uk/feedback/storage

Get the slides online

https://co572.pages.doc.ic.ac.uk/decks/Storage.pdf