Query Processing Models

Holger Pirk

Purpose of this lecture

By the end, you should

- Understand the principles of the different Query Processing Models
- Be able to implement queries according to them
- Understand their tradeoffs

Context

When does the right processing model matter?

- When the storage backend is fast
- Today, we are talking about main-memory databases
- It is going to become clear why this is important to keep in mind

In-memory Data Processing

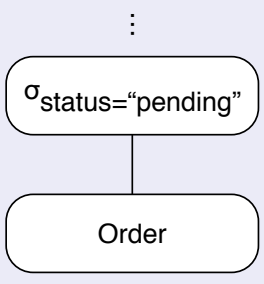
No more buffer management

- I told you, disk-based systems have their own buffer manager
  - remember:
    `readTupleFromPage("relationName", tupleID)[attributeID]`
- Many In-memory systems do not
  - (By the end of the lecture, you should understand why)
  - the new interface is `relationName[tupleID][attributeID]`
  - There are exceptions (of course)

But First: Let's look at some more operators

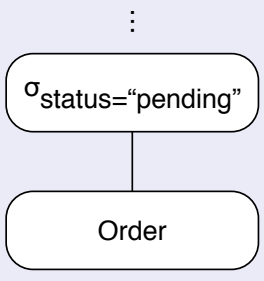
Single Selections

The Plan



Single Selections

The Plan

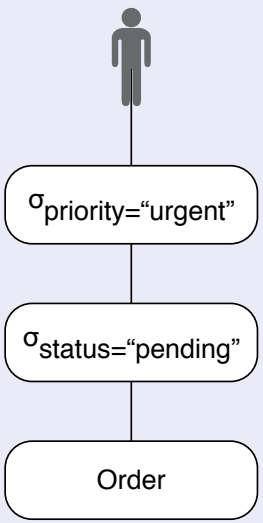


Implementation

```
for (size_t i = 0; i < orderSize; i++) {
 if(order[i].status == "pending")
  // here is where things get interesting:
  processTupleInNextOperator(inputRelation[i]);
 }
```

Selections in a Plan

The Plan



Implementation

```
for (size_t i = 0; i < orderSize; i++) {
 if(order[i].status == "pending")
  if(order[i].priority == "urgent")
   writeToOutput(inputRelation[i]);
 }
```

Group Bys (hash-based)

Implementation

```
// grouping by nation
// calculating count(*)
// we assume hashTable is initialized with zeroes & no zeroes in
    input

extern struct {int group; int value;}* hashTable;
for (size_t i = 0; i < inputSize; i++) {
 auto hashValue = hash(customer[i].nation);
 while (hashTable[hashValue].group &&
     hashTable[hashValue].group != customer.nation)
  hashValue = nextSlot(hashValue);
 hashTable[hashValue].group = customer[i].nation;
 hashTable[hashValue].count++;
 }
```

Pipeline Breakers

Definition

- A pipeline breaker is an operator that produces the first (correct) output tuple only after all input tuples have been processed

Quiz: Which of the relational operators are Pipeline Breakers

- Selections
- Projects
- Hash-Joins
- Sort-Merge-Joins
- Group By
- Order By
- Union
- Difference

Processing Models

What we know

- How to implement a join
- How to implement a selection
- How to implement a group by
- ...

What we don't know

- How to implement an entire query
  - With joins and selections and group by and them all
- The processing model is the strategy to combine operators

Volcano Processing - Your grandfather's processing model

Volcano Processing

Goals

- Flexibility
- Clean Design
- Maintainability
- Developer Prductivity

Function Objects

Call them Lambdas, Function pointers, etc.

- They are pieces of code that are treated like data
  - They are basically pointers to an instruction
- You can assign them to variables...
- ...you can pass them as parameters to other functions...
- ...and you can evaluate/invoke/call them with arguments...
- ...and they will return a value

C++ Syntax

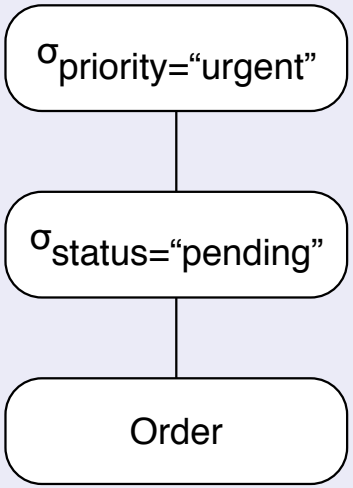
```
std::function<int(int, Tuple)> aggregate = [](auto x, auto y){
    return x+y;};
```

Python Syntax

```
aggregate = lambda x, y : x + y
```

Volcano Processing

The query



Framework

```
#include <functional>
class Tuple{
 //...
};
class Operator {
public:
 virtual Tuple next();
};
```

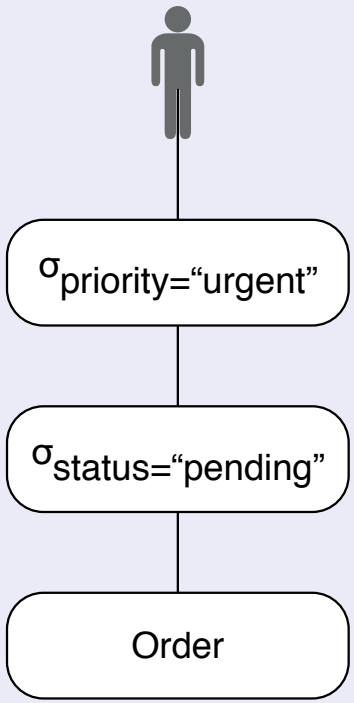
Operators

```
class Relation {
 std::vector<Tuple> r;
 int i = 0;
public:
 Tuple next(){
  return r[i++];
 };
};

class Select : public Operator{
public:
 Operator input;
 std::function<bool(Tuple)>
     predicate;
 Tuple next(){
  auto candidate = input.next();
  while(!predicate(candidate))
   candidate = input.next();
  return candidate;
 };
};
```

Volcano Processing Plan Creation

The query



Plans

```
Select({
  .predicate = [](Tuple a){ return a.status = "
      urgent"},
   .input = Select({
     .predicate = [](Tuple a){ return a.status =
         "pending"},
      .input = Relation({.r = loadRelation("
         Order")})})})})
```

Volcano Processing

Group By

```
class GroupBy : public Operator{
public:
 Operator child;
 std::function<int(int, Tuple)> aggregate;
 std::function<int(Tuple)> getGroupValue;
 Tuple* hashTable;
 int outputCursor = 0;

 void open(){
  auto inputTuple = child.next();
  while(inputTuple){
   auto hashValue = hash(getGroupValue(inputTuple));
   auto group = getGroupValue(inputTuple);
   while (hashTable[hashValue].group &&
       hashTable[hashValue].group != group)
    hashValue = nextSlot(hashValue);
   hashTable[hashValue].group = group;
   hashTable[hashValue].count = aggregate(hashTable[hashValue].
       count, inputTuple);
   inputTuple = child.next();
  }
 };
```

Calculating Buffer Pool I/O in Volcano

Scans

- Sequential I/O like we calculated in the Storage session

Pipeline Breakers: Input

- If buffer fits in memory: No I/O
- Otherwise: one page replacement per accessed tuple

Pipeline Breakers: Output

- If buffer fits in memory: No I/O
- Otherwise: sequential I/O over output tuples

All others

- Cause no I/O

Worksheet: Volcano Processing

Customers

- 10.000 Tuples
- attributes: `id, name, address, nation, phone, accountNumber`
- strings are dictionary-compressed

Orders

- 5 Million Tuples
- attributes: `date, status, priority, discount`
- strings are dictionary-compressed

Buffer Pool

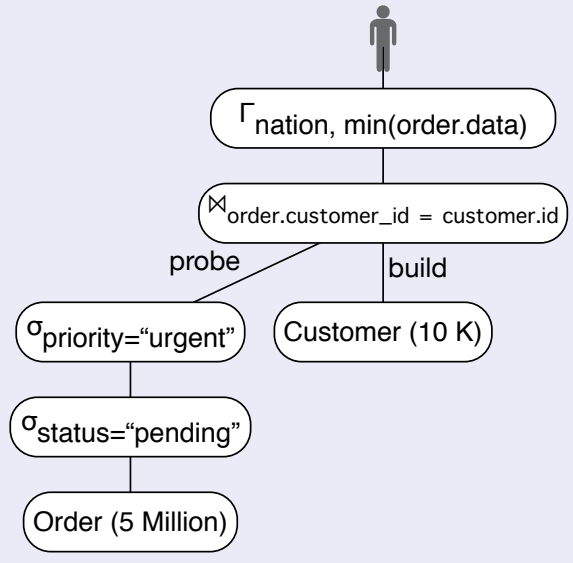
- Is the Level 2 CPU Cache
  - 512 KB, LRU
  - organized in 64 Byte Pages (called cache lines)

Worksheet: Volcano Processing

Consider the following query

```
select nation, min(order.date)
from orders join customer
using (orders.customer_id = customer.id)
where order.status = "pending"
and order.priority = "urgent"
group by nation
```

Query Plan



Task

- Calculate the number of Page Faults/Cache Misses
  - Selectivities: 50% each
  - N-ary, spanned pages, 10K Customers, 5M Orders, 193 Countries
  - 512 KB Cache, 64 Byte Pages, 2 $\times$ overallocated hashtables

CPU Efficiency

What is the cost of a (sequential) memory access

- Back of the envelope calculation
  - My MacBook has 37.5 GB/s memory bandwidth and 4 cores @ 2.9 GHz
  - 9.375 GB/s per core
  - 3.23 Bytes per cycle (let's say about one integer)
- We better make sure we can process one integer per cycle

Function pointers

How they are evaluated by a CPU (roughly)

- The CPU stores the current instruction pointer (the `call`)
- The arguments are put on the execution stack
- The CPU instruction pointer is set to the address of the first instruction of the function code
  - This is called a Jump (JMP)
- The function is executed until it returns (there is a special return instruction)
- The instruction pointer is set to the instruction after the `call`

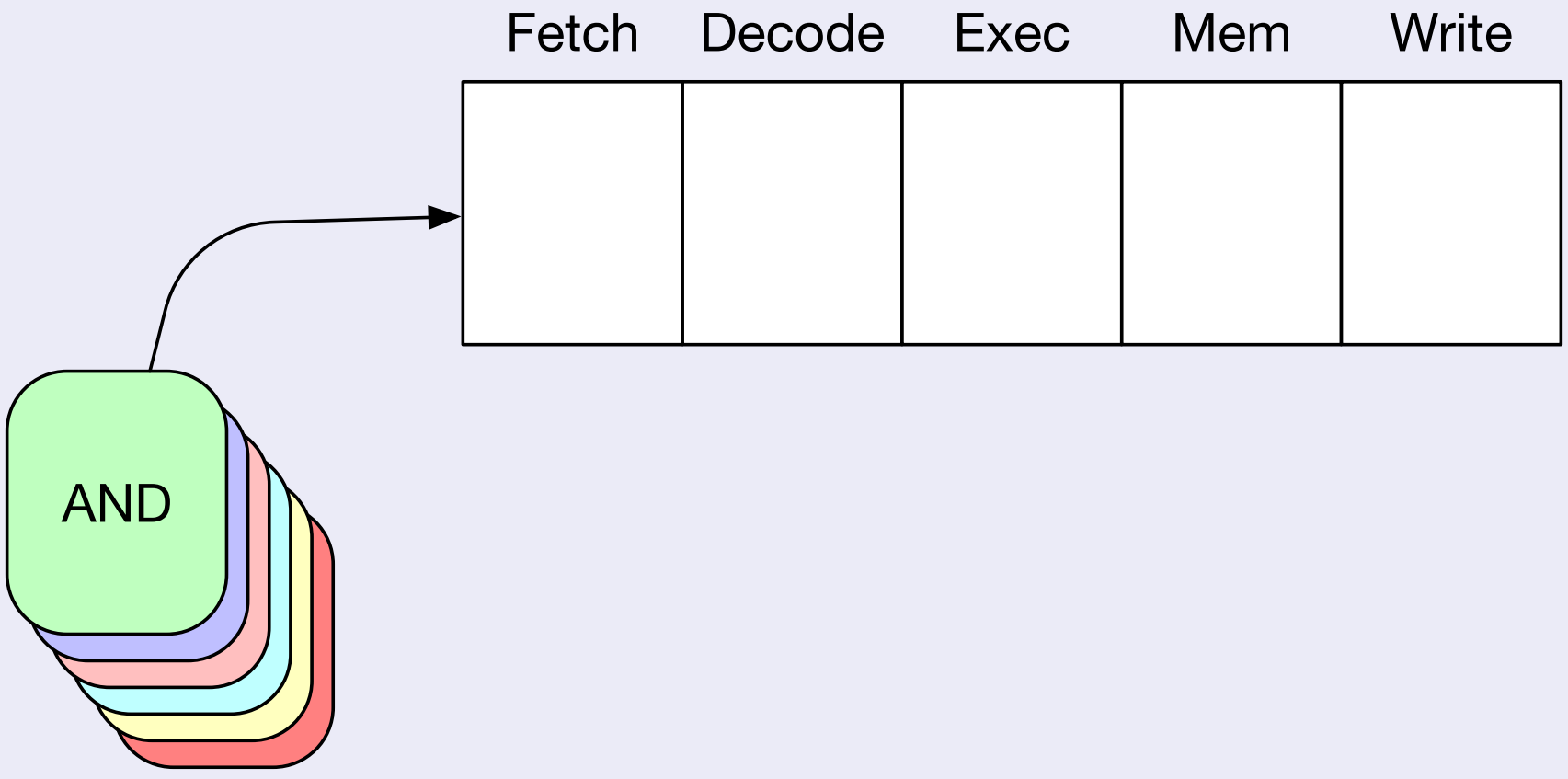
CPU execution pipelining (simplified)

Modern CPUs...

- ...execute instructions in stages...
- ...like `fetch`, `decode`, `execute`, `memory read`, `write result`.
- Instructions spend one cycle in each stage...
- ...and move on to the next stage after every cycle.

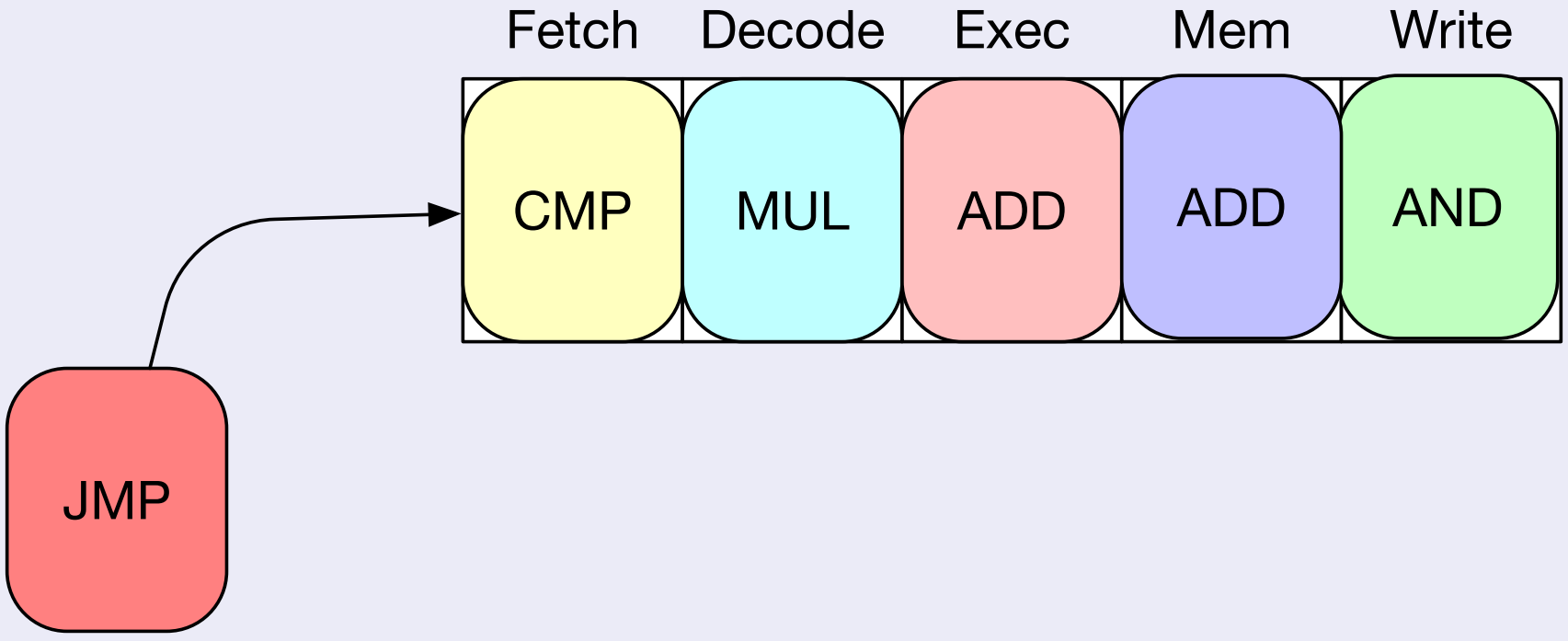
CPU execution pipelining (simplified)

An Empty CPU Pipeline



CPU execution pipelining (simplified)

A Filled CPU Pipeline

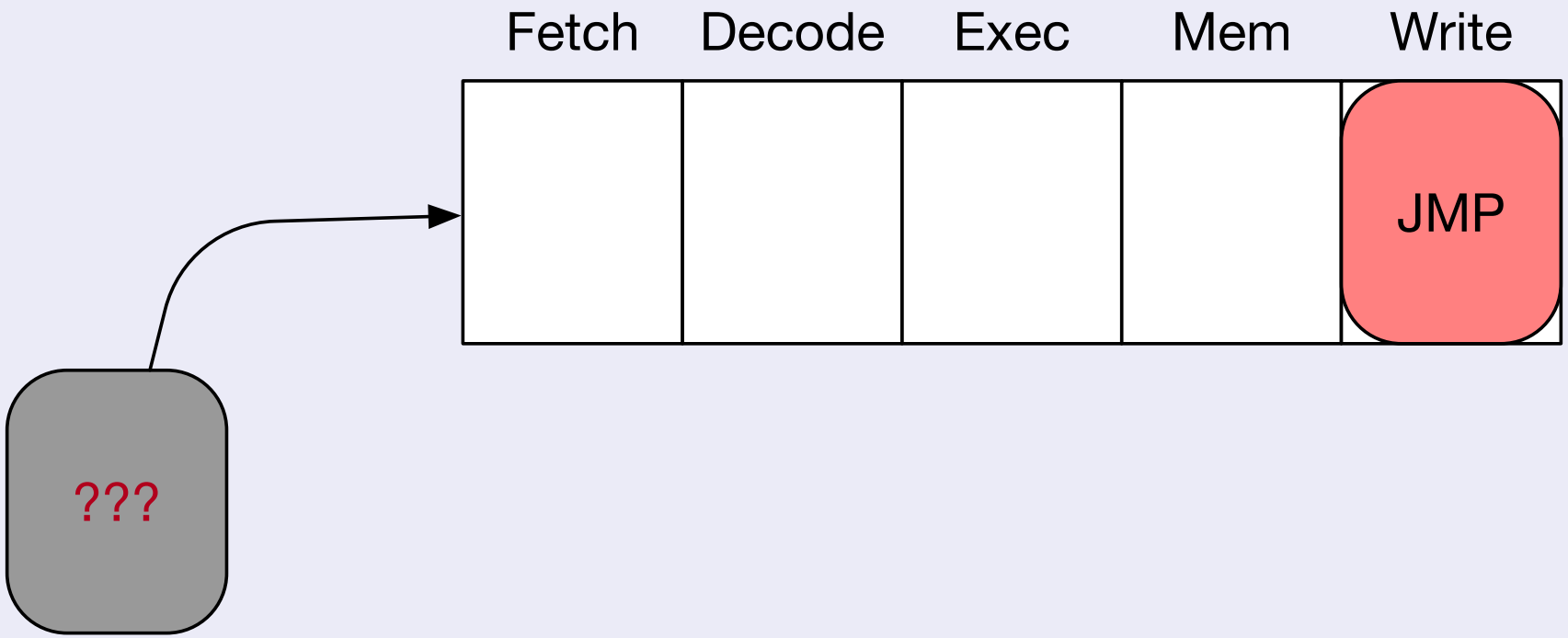


Function Pointers cause Pipeline Bubbles

Pipeline Bubbles (the technical term is Control Hazard)

- Remember: a Jump sets the instruction pointer to an arbitrary address
- The CPU needs to read the next instruction from this address
- Ergo: the next instruction can only be read once the jump is complete

A Control Hazard



The cost of a pipeline bubble

Impact

- Dependent on the length of the pipeline
  - My Macbook's CPU has around 15 stages

Bulk Processing

How many Function calls are there?

Per-tuple...

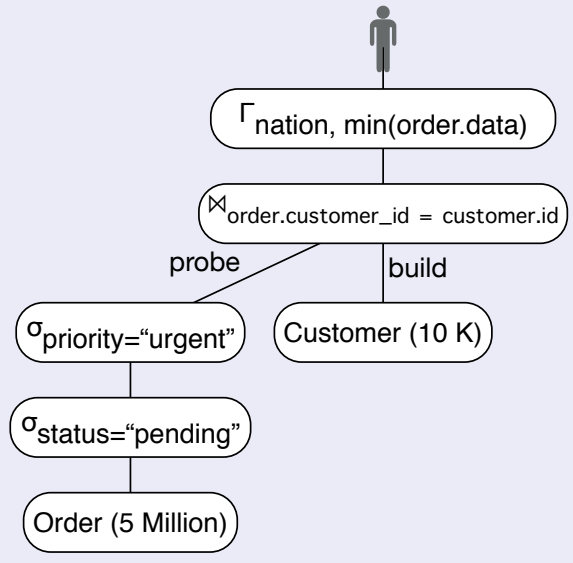
Selections	One to read the input, one to apply the predicate
Hash-Join Build	One to read the input
Hash-Join Probe	One to read the input
Group-By	One to read the input, one to extract the group key, one to calculate the new aggregate

Worksheet: CPU Costs

Consider the following query

```
select nation, min(order.date)
from orders join customer
using (orders.customer_id = customer.id)
where order.status = "pending"
and order.priority = "urgent"
group by nation
```

Query Plan



Task

- Calculate the number of function calls
  - Selectivities: 50% each
  - N-ary, spanned pages, 10K Customers, 5M Orders, 193 Countries
  - 512 KB Cache, 64 Byte Pages, 2 $\times$ overallocated hashtables

The Bottom Line

What factor is bounding performance?

Bulk Processing

Bulk Processing

The problem

- If CPU is the bottleneck...
- ...and function calls dominate CPU costs...
- ...can we process queries without any function calls
  - (or at least as few as possible)

The idea

- Turn *Control Dependencies* into *Data Dependencies*:
  - Instead of processing tuples right away, buffer them
  - Fill the buffer with lots of tuples
  - Pass the buffer to the next operator

What Bulk Processing looks like

A Bulk Processing Program

```
class Tuple{};
int orderSize;
Tuple* order, buffer1, buffer2;

int selectStringCompare(Tuple* outputBuffer, Tuple* inputBuffer,
    int inputSize, string predicate, int attributeOffset){
 auto outputCursor = 0;
 for (size_t i = 0; i < inputSize; i++) {
  if(*((string*)inputBuffer[i] + attributeOffset) == predicate)
   outputBuffer[outputCursor++] = inputBuffer[i];
 }
 return outputCursor;
};

auto buffer1Size = selectStringCompare(buffer1, order, orderSize, "
    pending", 1);
auto buffer2Size = selectStringCompare(buffer2, buffer1,buffer1Size
    , "urgent", 2);
```

Bulk Processing

Bulk Processing means tight loops

- No function calls, no jumps
- Very CPU efficient
- Free Insight: Bulk Processing is like making every operator a pipeline breaker

However,

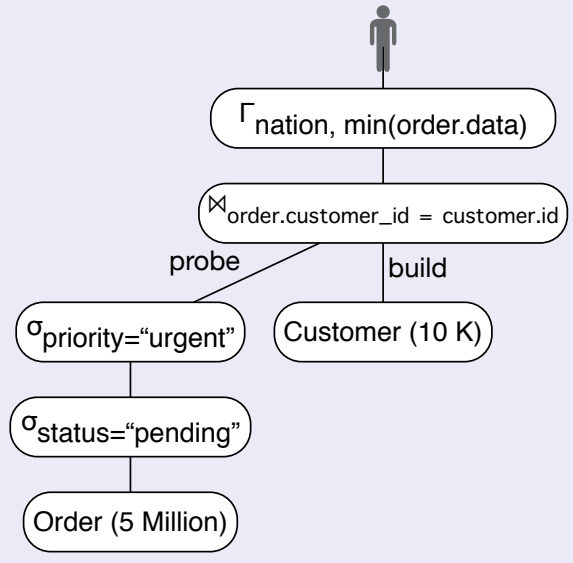
- We are materializing intermediate results...
  - ...and aren't databases I/O bound?
  - ...well, are they?

Worksheet: Bulk Processing

Consider the following query

```
select nation, min(order.date)
from orders join customer
using (orders.customer_id = customer.id)
where order.status = "pending"
and order.priority = "urgent"
group by nation
```

Query Plan



Task

- Calculate the number of Page Faults/Cache Misses
  - Selectivities: 50% each
  - N-ary, spanned pages, 10K Customers, 5M Orders, 193 Countries
  - 512 KB Cache, 64 Byte Pages, 2 $\times$ overallocated hashtables

Bulk Processing and Decomposed Storage

By-Reference Bulk Processing

Saving Bandwidth

- We are copying a lot of data around
  - This is a classic computing problem with a classic solution:
  - Call by reference
- Instead of producing tuples, we produce their IDs (positions in their buffer)
- When processing a tuple, we always use the ID to look up the actual value
  - Lookup costs are the same as they are for a hashtable without conflicts

By-Reference Bulk Processing

Code

```
 class Tuple{};
 int orderSize;
 Tuple* order;
 int* buffer1, buffer2;

int selectStringCompare(Tuple* outputBuffer, int* inputBuffer, int
    inputSize,
             string predicate, int attributeOffset, Tuple*
                 underlyingRelation){
  auto outputCursor = 0;
  if(!inputBuffer){
   for (size_t i = 0; i < inputSize; i++) {
    if(*((string*)underlyingRelation[i] + attributeOffset) ==
        predicate)
     outputBuffer[outputCursor++] = i;
   }
  } else {
   for (size_t i = 0; i < inputSize; i++) {
    if(*((string*)underlyingRelation[inputBuffer[i]] +
        attributeOffset) == predicate)
     outputBuffer[outputCursor++] = inputBuffer[i];
   }
```

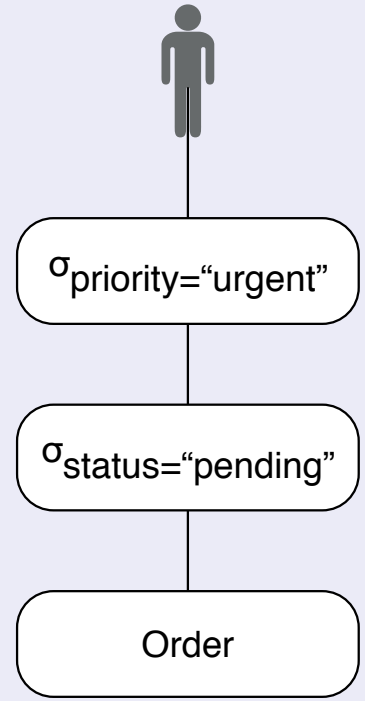
Worksheet: By-Reference Bulk Processing

Consider the following query

```
select count(*) from orders
where order.status = "pending"
and order.priority = "urgent"
group by nation
```

- Order:
  `date, status, priority, discount`
- Customer:
  `id, name, address, nation, phone, ac`

Query Plan



Task

- Calculate the number of Page Faults/Cache Misses
  - Selectivities: 50% each
  - N-ary, spanned pages, 10K Customers, 5M Orders, 193 Countries
  - 512 KB Cache, 64 Byte Pages, 2 $\times$ overallocated hashtables

Calculating Page Access probability from selectivity

How to do it

- Selectivity s is the percentage of tuples being touched
- Assume uniformly distributed values
- Assume n tuples on a page
- What is the probability of any one of them being touched?
  - $1 - (1 - p)^n$

By-Reference Bulk Processing of Decomposed Data

Saving even more bandwidth

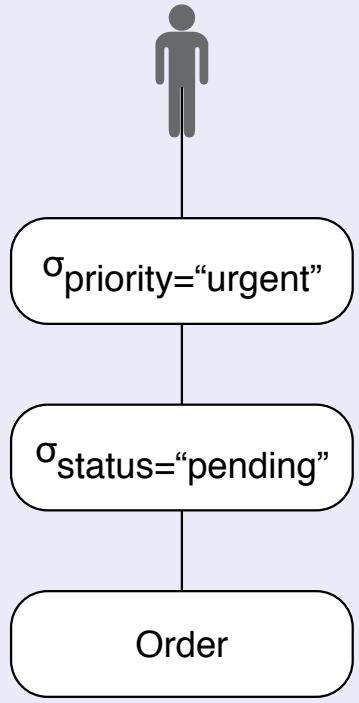
- Every operator processes exactly one column of a tuple
- In N-ary, storage, values of a tuple are co-located on a page
  - i.e., you always pay for all values on a page (even if you only process one)
  - these useless values also occupy space in the buffer pool/cache
- DSM fixes both of these problems
  - DSM was introduced to databases as a consequence of Bulk Processing
  - Not the other way around
  - Trust me, I know!

Worksheet: By-Reference Bulk Processing of DSM Data

Consider the following query

```
select count(*) from orders
where order.status = "pending"
and order.priority = "urgent"
group by nation
```

Query Plan



Task

- Calculate the number of Page Faults/Cache Misses
  - Selectivities: 50% each
  - DSM, spanned pages, 10K Customers, 5M Orders, 193 Countries
  - 512 KB Cache, 64 Byte Pages, 2 × overallocated hashtables

The End