Storage

Holger Pirk

Purpose of this lecture

We are going down the rabbit hole now

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

Understand storage alternatives

- N-ary vs. decomposed storage
- in-memory vs. disk
- slotted pages

Connection

Using an ORM to insert data

Example C++ code using ODB

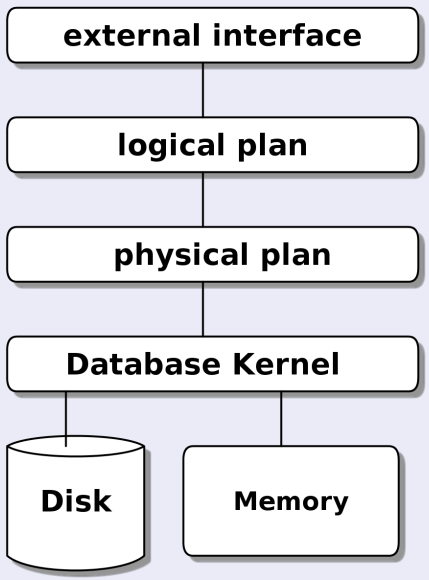
```
transaction t(db.begin());
Employee me;
me.payrollNumber = 4;
me.name = "holger";
me.salary = 100*1000;
me.joiningDate = 43429342;
db.persist(me);
t.commit();
```

In SQL

```
INSERT INTO "Employee" VALUES(4,'holger',100000,43429342);
```

Recall:

Database Architecture



Insert processing

- Inserts are usually not optimized
- Arrive at the storage layer as intact tuples

Processing tuples

Where can we put them?

- Disk
- Memory
- technically, there are many alternatives

Let's talk about memory first

- Mostly because it is less involved

N-ary vs. Decomposed Storage

How to store tuples

Another mismatch

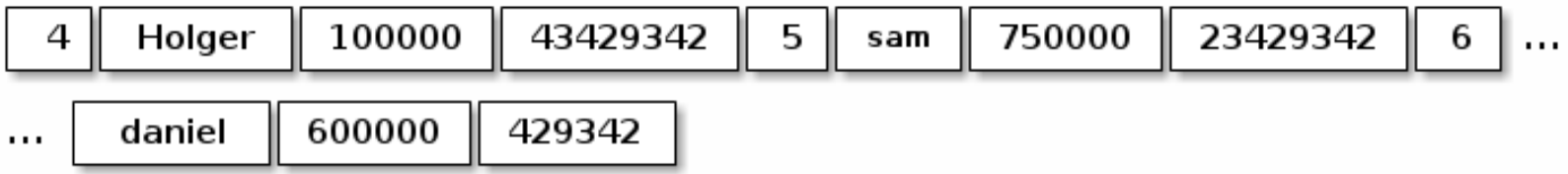
- Relations are two-dimensional
- Memory is one-dimensional
- Tuples need to be linearized
- There are two mainstream strategies
  - (and research on hybrids)

The N-ary Storage Model (NSM)

Example

```
(4,'holger',100000,43429342);
(5,'sam',750000,23429342);
(6,'daniel',600000,13429342);
```

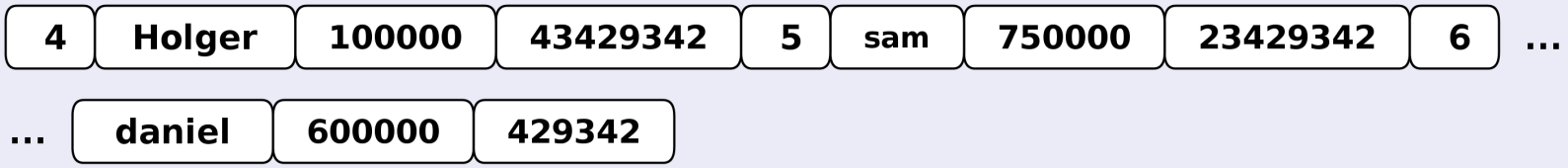
Linearization in N-ary format



- In marketing they call this a row store

When N-ary storage works well

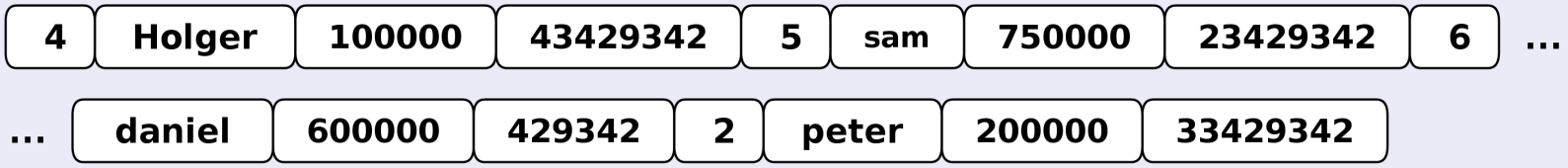
Linearization in N-ary format



Insert a new tuple

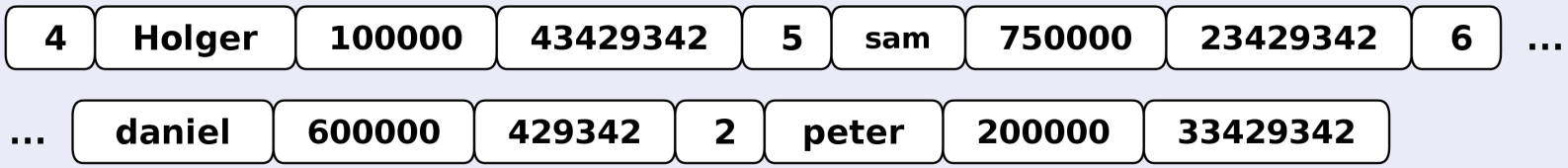
```
(2,'peter',200000,33429342);
```

Simply append the tuple



When N-ary storage is suboptimal

The database



They query

- calculate the sum of all salaries
- In SQL `select sum(salary) from employee`

The Decomposed Storage Model (DSM)

Example

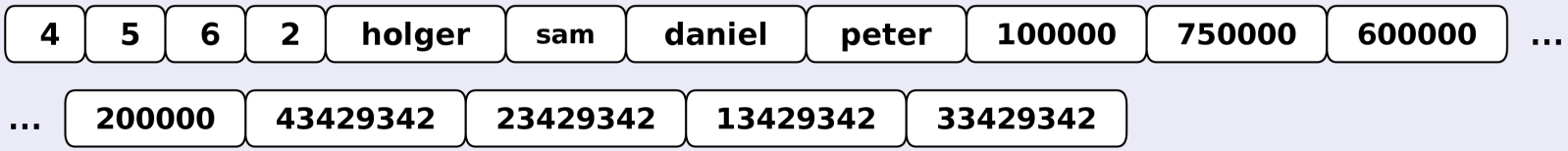
```
(4,'holger',100000,43429342);
(5,'sam',750000,23429342);
(6,'daniel',600000,13429342);
```

Linearization in Decomposed format

- In marketing they call this a column store

When decomposed storage works well

The database



They query

- calculate the sum of all salaries
- In SQL `select sum(salary) from employee`

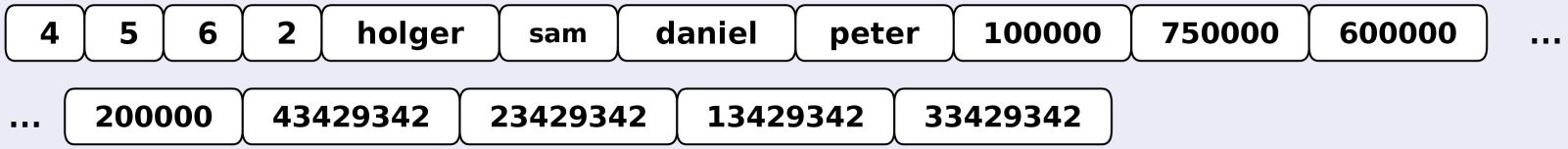
When decomposed storage is supoptimal

Linearization in Decomposed format

Insert a new tuple

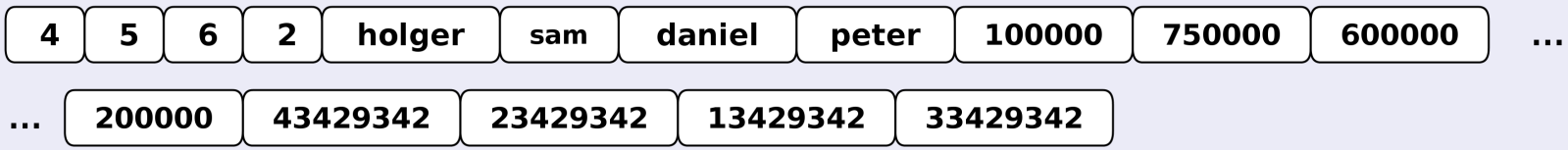
```
(2,'peter',200000,33429342);
```

Tuples need to be decomposed before insertion



When decomposed storage is supoptimal

Tuples are decomposed in database



Access a single tuple

```
select * from employee where tuple_id = 3; -- assume index access
```

- tuple needs to be reconstructed

Impact of NSM vs. DSM

Locality is king

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

Impact of NSM vs. DSM

Locality is king

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

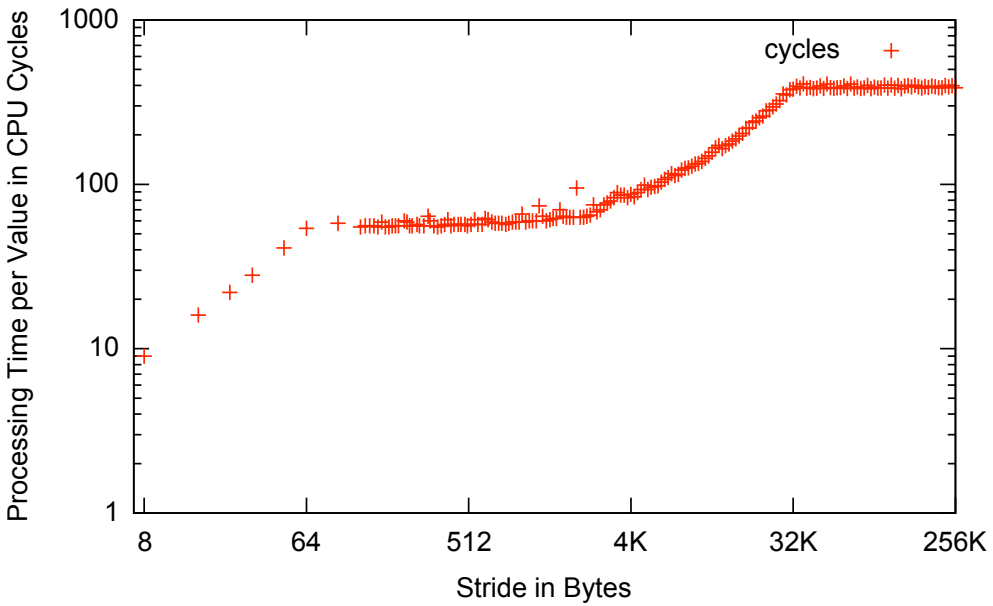
How much locality is necessary

Impact of NSM vs. DSM

Locality is king

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

How much locality is necessary

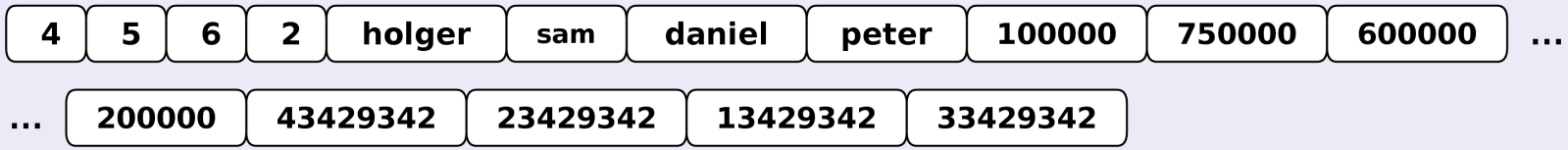



```
for (size_t i = 0; i < N; i++)
  sum += input[i*stride];
```

Variable Sized Datatypes

Let's think back to our example

Linearization in Decomposed format

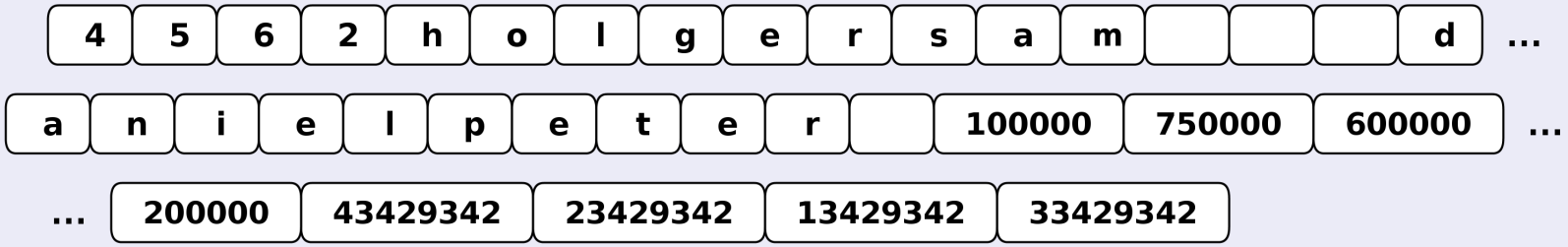


Names have different sizes

- $\rightarrow$ they occupy variable space in memory
- We'd like to maintain fixed tuple sizes,
  - allows randomly access to tuples by their position
  - e.g., when reconstructing decomposed tuples
- Two choices
  - overallocate space for varchars (many systems need a size parameter: e.g., `varchar(128)`)
  - store them out of place

In place storage

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

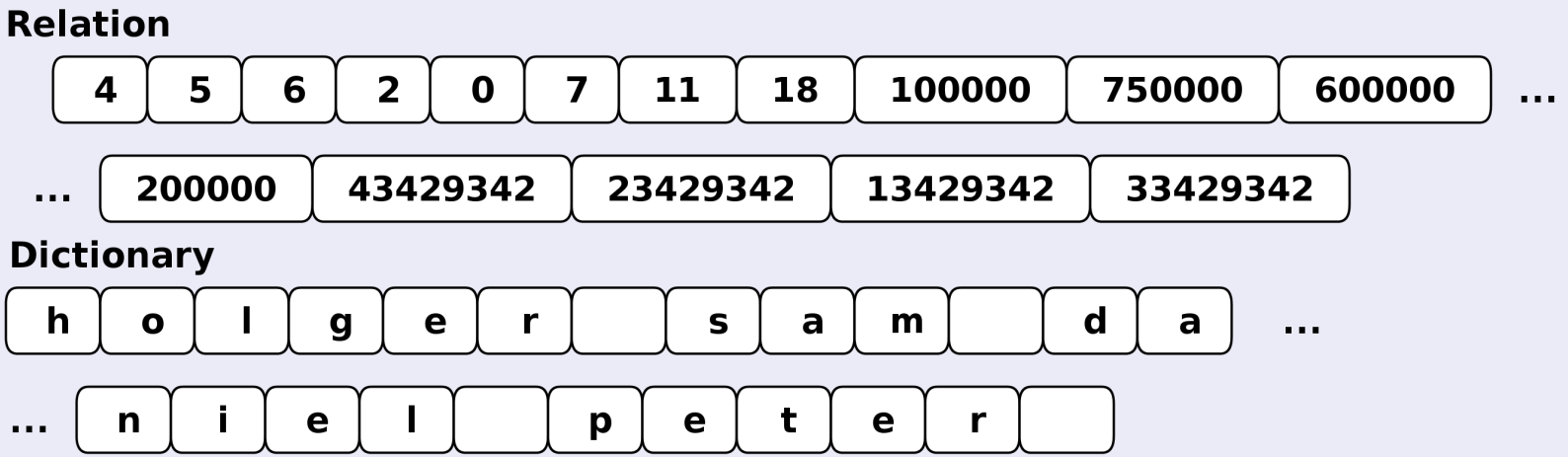


Properties

- 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



Properties

- Space conservative
- Bad for locality
- Complicated
  - Harder to implement
  - Garbage-collection is tricky

Worksheet

Assume the following

Query Find the names of all employees with a salary above 700.000

Costs Each (random) access to a value costs 3, each access to an imediately following value costs 1, accessing a value again is free

On the following database

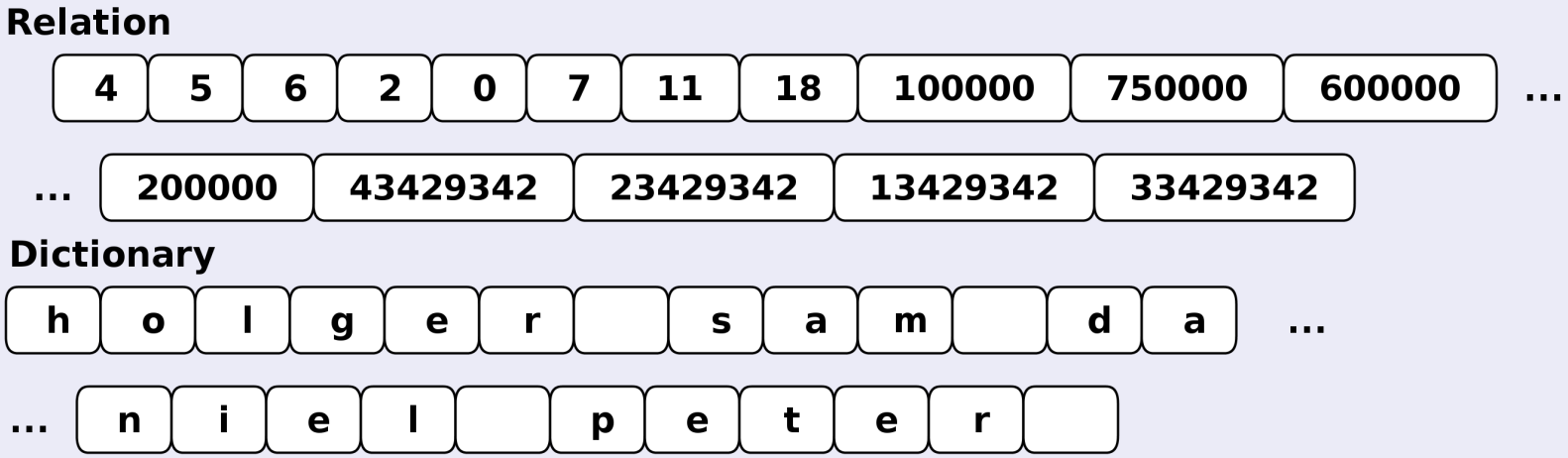
```
(4,'holger',100000,43429342);
(5,'sam',750000,23429342);
(6,'daniel',600000,13429342);
(2,'peter',200000,33429342);
```

Calculate the data access costs on these storage strategies! Which is the least expensive?

- Out-of-place N-ary storage
- In-place N-ary storage
- Out-of-place Decomposed storage
- In-place Decomposed storage

Nifty out of place storage: dictionary compression

Our example database stored in out of place DSM



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

Worksheet

Assume the following

Query Find the names of all employees with a salary above 700.000

Costs Each (random) access to a value costs 3, each access to an imediately following value costs 1, accessing a value again is free

On the following database

```
(4,'holger',100000,43429342);
(5,'sam',750000,23429342);
(6,'daniel',600000,13429342);
(2,'peter',200000,33429342);
(9,'sam',920000,13919390);
```

Calculate the data access costs on these storage strategies! Which is the least expensive?

- Out-of-place N-ary storage with duplicate elimination
- In-place N-ary storage
- Out-of-place Decomposed storage with duplicate elimination
- In-place Decomposed storage

Data storage on disk

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

Goals shift

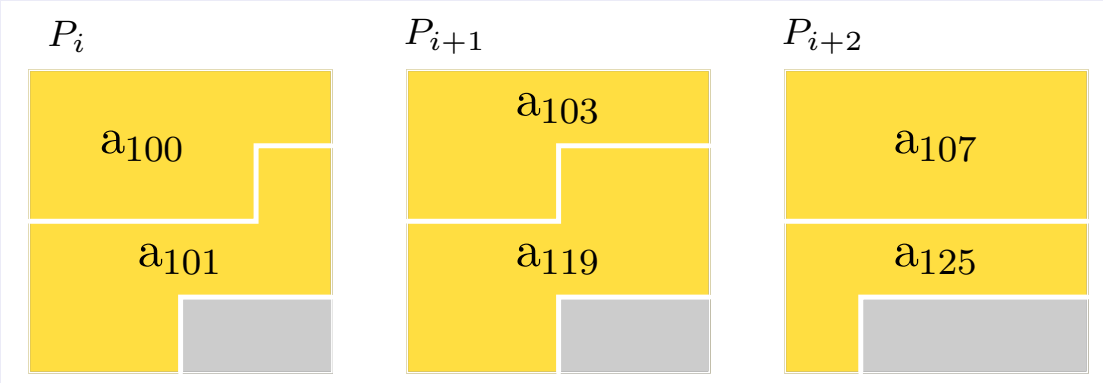
- Disks are dominating costs by far
  - Complicated I/O management strategies pay off
- Pages are large
  - Each page behaves like a mini-database
  - (in the case of N-ary storage)

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



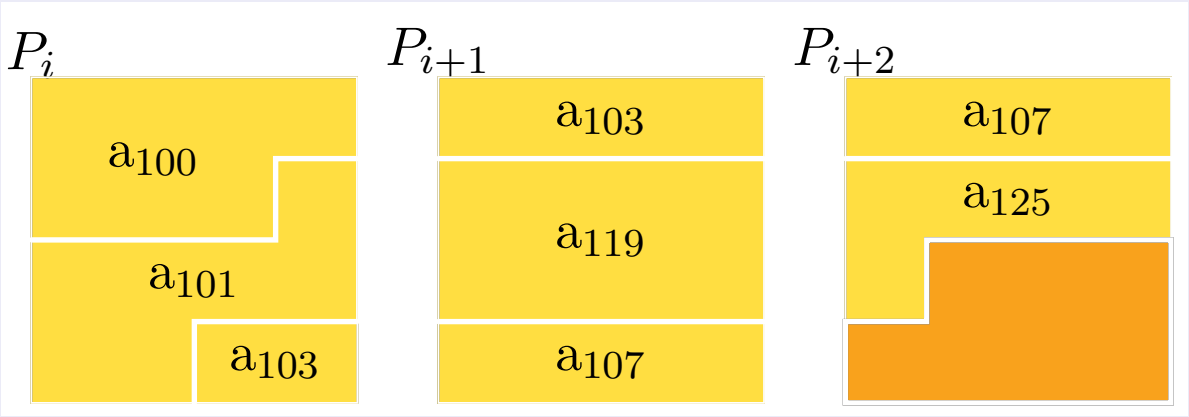
- Disadvantage:
  - Space waste
  - Cannot deal with large records
  - No in-page random access for variable-sized records

Spanned Pages

Goals

- Minimize space waste
- Support large records

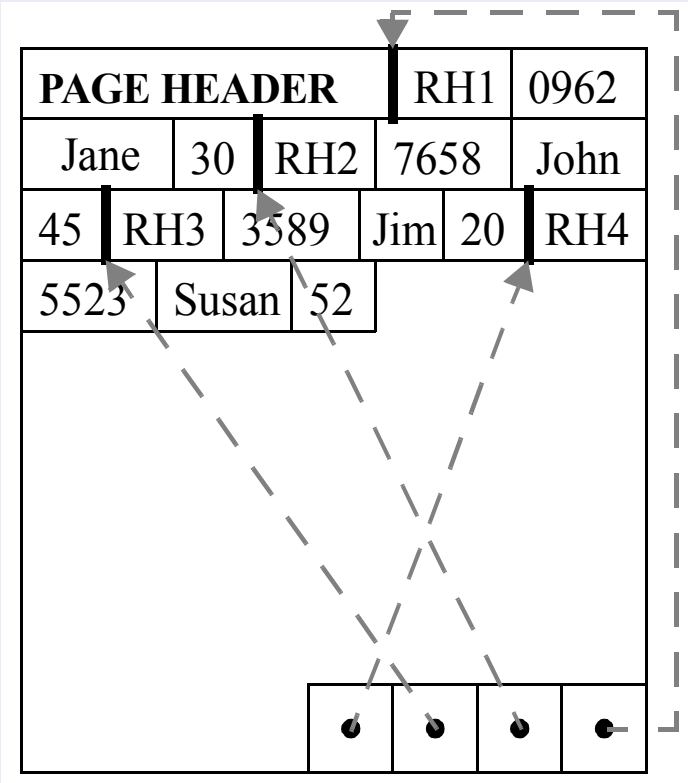
Example



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

Slotted Pages

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

Quiz

In a particular database, each record occupies 620 bytes, and records are stored in in-place, spanned pages of 2KB.

- How many pages are required to stored 100 records?
  - 30
  - 31
  - 33
  - 34

Quiz

Assume a database with records of size 17 bytes, stored in unspanned, slotted 4KB pages. How many tuples can be stored in 10 blocks?

- 2150
- 2155
- 2156
- 2160
- 2400
- 2410

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

The End