Trường ĐH Bách Khoa Tp.HCM
Khoa Khoa Học và Kỹ Thuật Máy Tính

Physical Database Design & Tuning

GV: PGS.TS ĐẶNG TRẦN KHÁNH

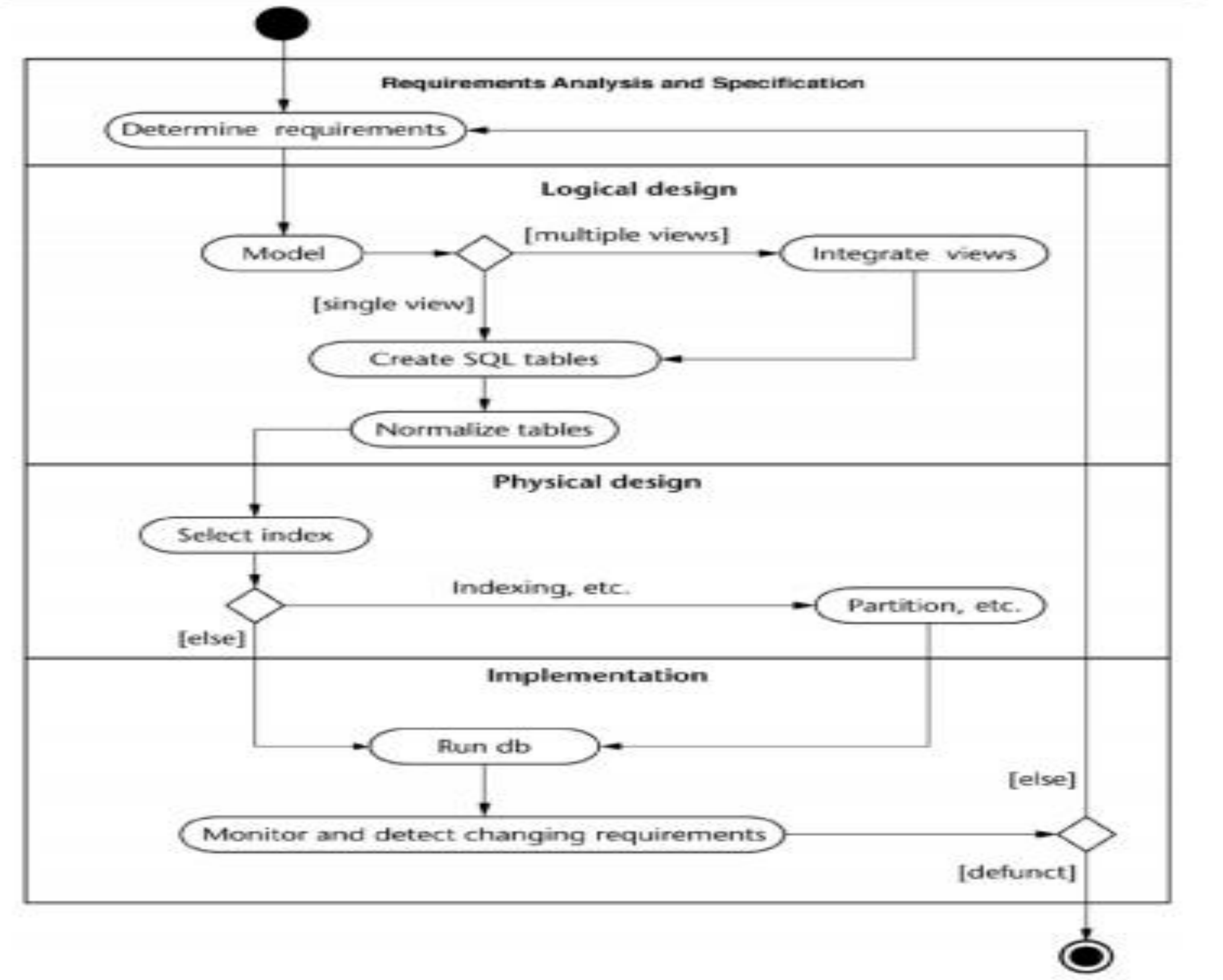
GROUP 3

Chu Xuân Tình - 1870583
Huỳnh Nguyễn Viết Thành - 1870578

Outline

1. introduction & Basic index methods
    1.1. B+ tree
    1.2. Compose index
    1.3. Bitmap index
    1.4. Summary
2. Selecting Materialized Views
    2.1. Simple View Materialization
    2.2. Exploiting Commonality
    2.3. Exploiting grouping and Generalization
    2.4. Resource Considerations
    2.5. Tips and Insights for Database Professionals

Relational DB design



Relational DB design

Database design phases:

(a) Requirement Analysis,

(b) Conceptual design

(c) Logical design

(d) Physical design

Concept:

- ✓ It is the process of transforming a logical data model into a physical model of a database
- ✓ It focuses on the methods of storing and accessing those tables on disk that enable the database to operate with high efficiency.

Physical Database Design

Physical Design Goal

- Physical Design Goal: definition of appropriate storage structures for a specific DBMS, to ensure the application performance desired
- Good design and tuning requires understanding the database **workload**.

  A *workload description* contains
  - the most important queries and their frequency
  - the most important updates and their frequency
  - the desired performance goal for each query or update

Physical Database Design

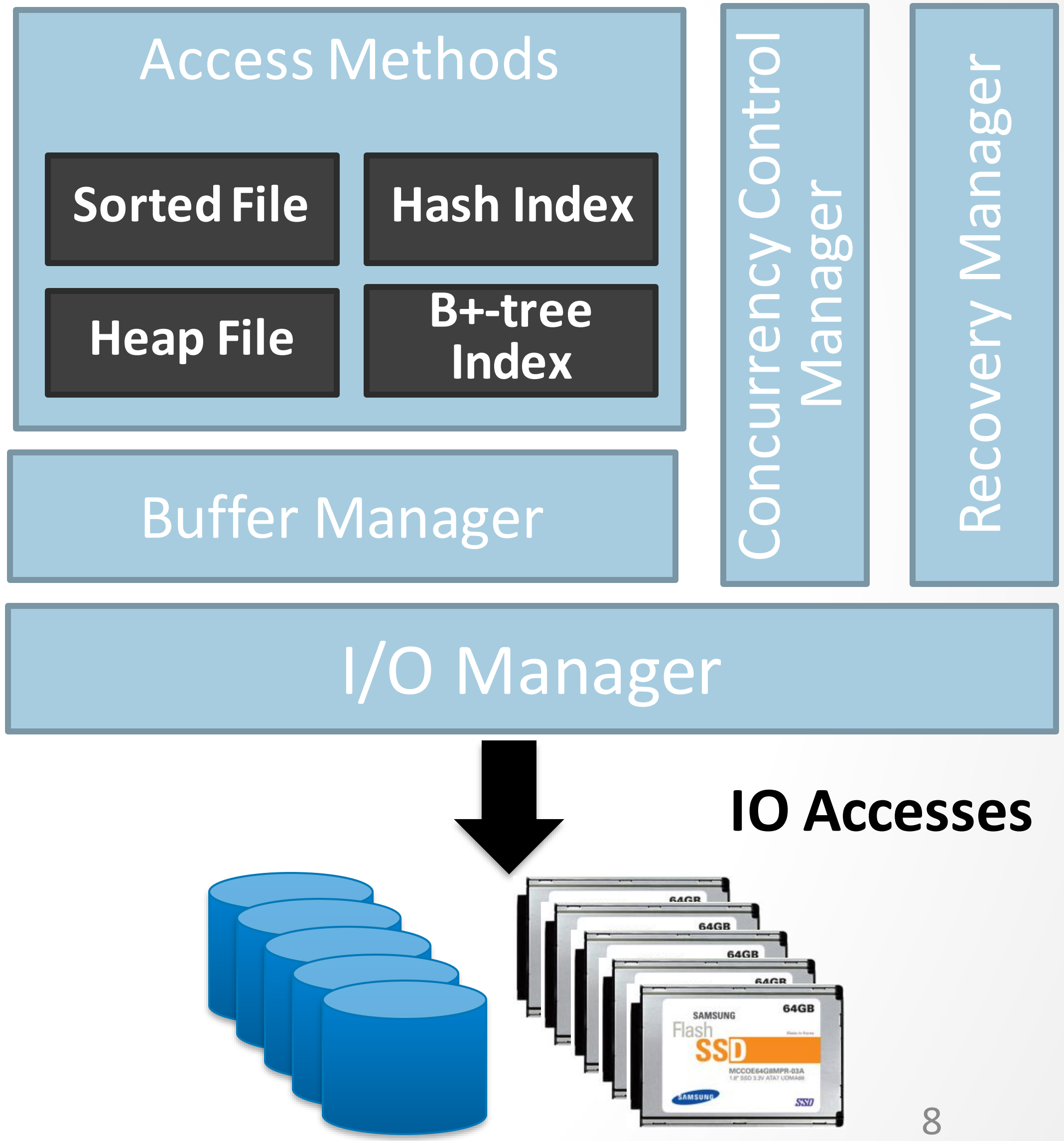
The physical database design step involves the selection of

- ❖ Indexes: An index is a data organization set up to speed up the retrieval (query) of data from tables.
- ❖ Partitioning:
  - ✓method for reducing the workload on any hardware component
  - ✓balance the workload across the system and preventing bottlenecks.
- ❖ Clustering:
  - ✓Technique by which data can be clustered by dimensions (such as location, timeframe or product type
  - ✓Offer potentially huge performance improvement for query processing by reducing the I/O processing
- ❖ Selective materialization of data

Physical Database Design

A storage strategy?

- ❖ Storage structures for relations:
  – heap (small data set, scan operations, use of indexes): Set of records, partitioned into blocks - Unsorted
  – sequential (sorted static data)
  – hash (key equality search), usually static
  – tree (index sequential) (key equality and range search)
- ❖ Choice of secondary index, considering that
  – they are extremely useful
  – slow down the updated of the index keys
  – require memory



Physical Database Design

Operations on an Index

- ❖ Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.

- ✓ Search: Quickly find all records which meet some *condition on the search key attributes*
- ✓ Insert / Delete entries

Indexing is one the most important features provided by a database for performance

Physical Database Design

Index Classification

❖ **Clustered/unclustered**

– Clustered = records close in index are close in data

– Unclustered = records close in index may be far in data

❖ **Primary/secondary**

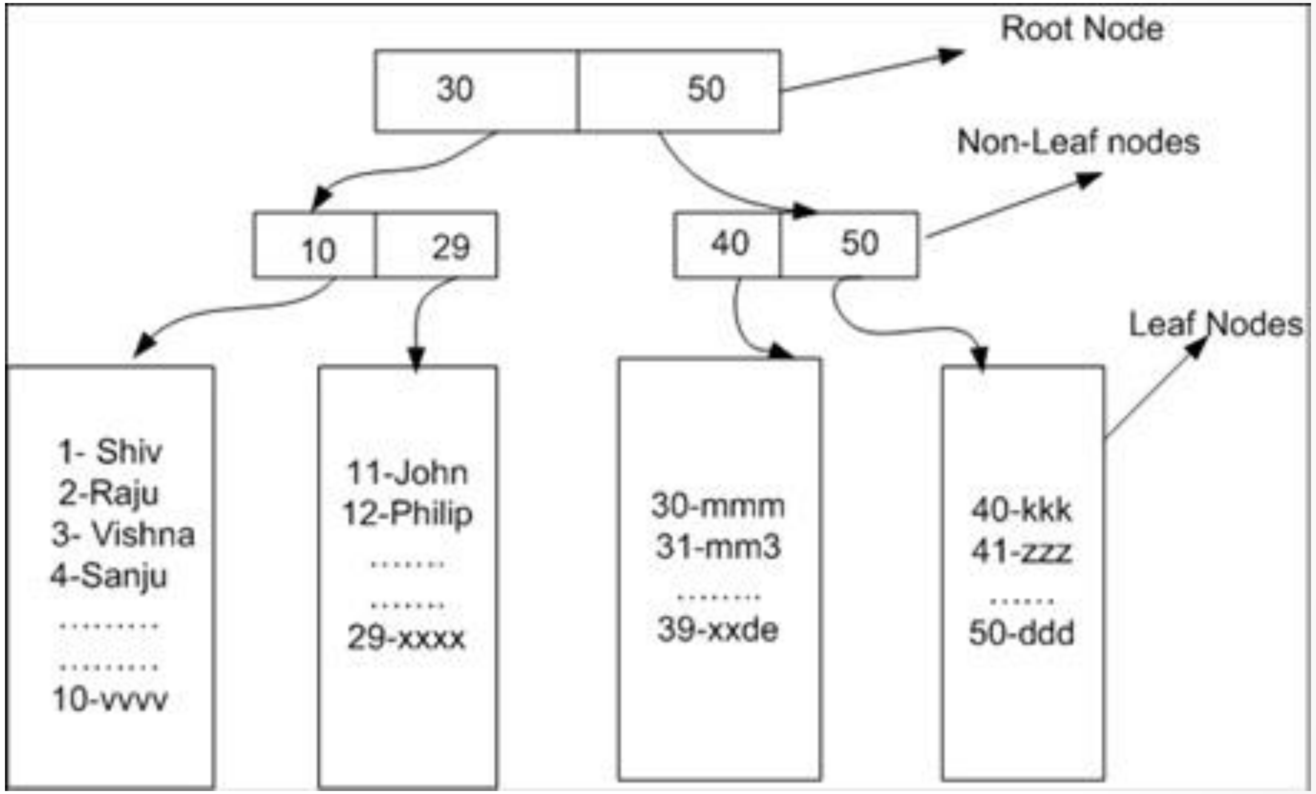
- Primary = is over attributes that include the primary key Secondary = otherwise

❖ **Multilevel Indexing**: B+ tree

Physical Database Design

Clustered

- Cluster index is a type of index which sorts the data rows in the table on their key values.
- A clustered index defines the order in which data is stored in the table
- Only one clustered index per table.



Physical Database Design

Clustered

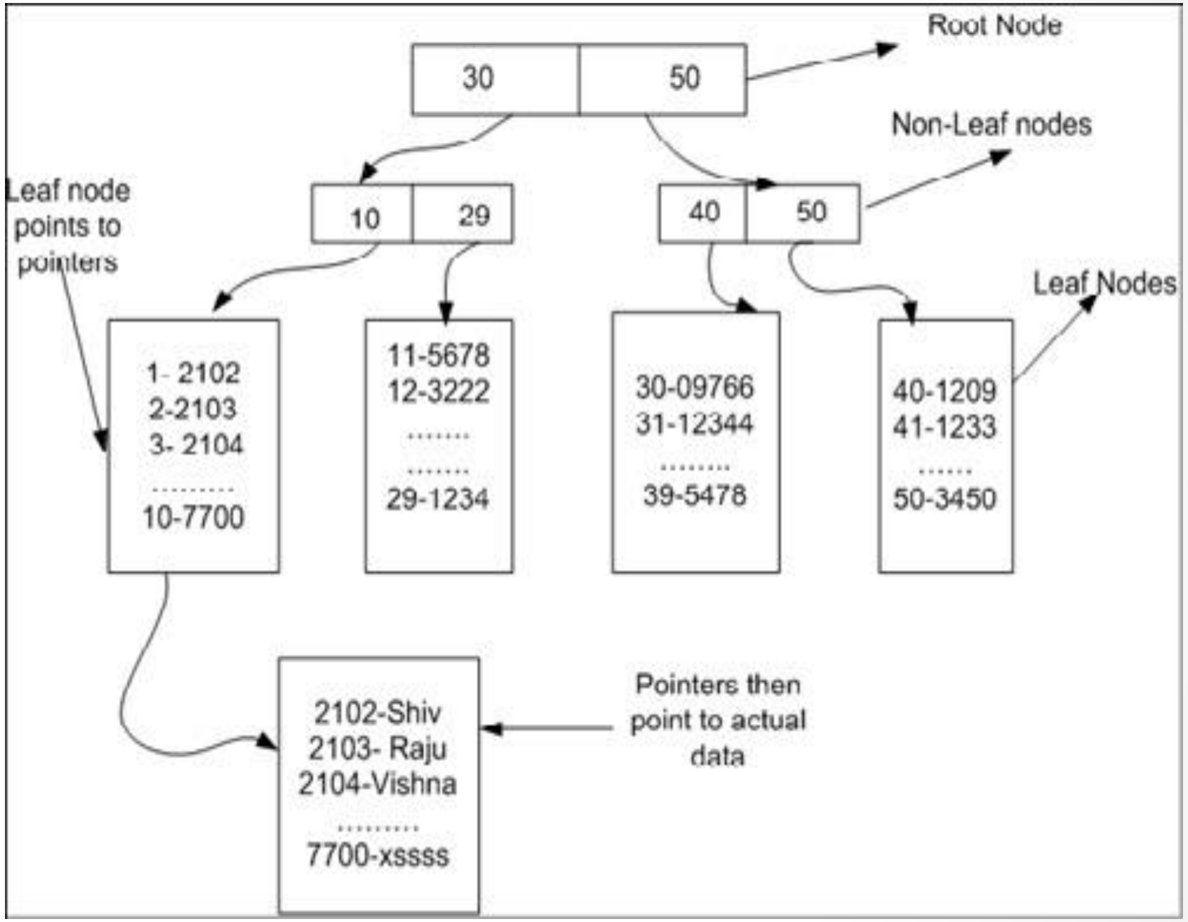
❖ **Advantages of Clustered Index**

- ✓ Clustered indexes are an ideal option for range or group by with max, min, count type queries
- ✓ In this type of index, a search can go straight to a specific point in data so that you can keep reading sequentially from there.
- ✓ Clustered index method uses location mechanism to locate index entry at the start of a range.
- ✓ It is an effective method for range searches when a range of search key values is requested.
- ✓ Helps you to minimize page transfers and maximize the cache hits.

Physical Database Design

Non-clustered index

- A Non-clustered index stores the data at one location and indices at another location
- The index contains pointers to the location of that data
- A single table can have many non-clustered indexes
- A non-clustering index is defined in the non-ordering field of the table



Physical Database Design

Non-clustered index

Advantages of Non-clustered index

- ✓ A non-clustering index helps you to retrieves data quickly from the database table.
- ✓ Helps you to avoid the overhead cost associated with the clustered index
- ✓ A table may have multiple non-clustered indexes in RDBMS. So, it can be used to create more than one index.

Physical Database Design

Clustered Index vs Non-Clustered Index

Parameters	Clustered	Non-clustered
Use for	You can sort the records and store clustered index physically in memory as per the order.	A non-clustered index helps you to creates a logical order for data rows and uses pointers for physical data files.
Storing method	Allows you to stores data pages in the leaf nodes of the index.	This indexing method never stores data pages in the leaf nodes of the index.
Size	The size of the clustered index is quite large.	The size of the non-clustered index is small compared to the clustered index.
Data accessing	Faster	Slower compared to the clustered index
Additional disk space	Not Required	Required to store the index separately
Type of key	By Default Primary Keys Of The Table is a Clustered Index.	It can be used with unique constraint on the table which acts as a composite key.
Main feature	A clustered index can improve the performance of data retrieval.	It should be created on columns which are used in joins.

B+ Tree

B+ Tree

- DBMS offer a variety of special access structures to find the requested record without reading the entire file. The most common one is the B+-tree

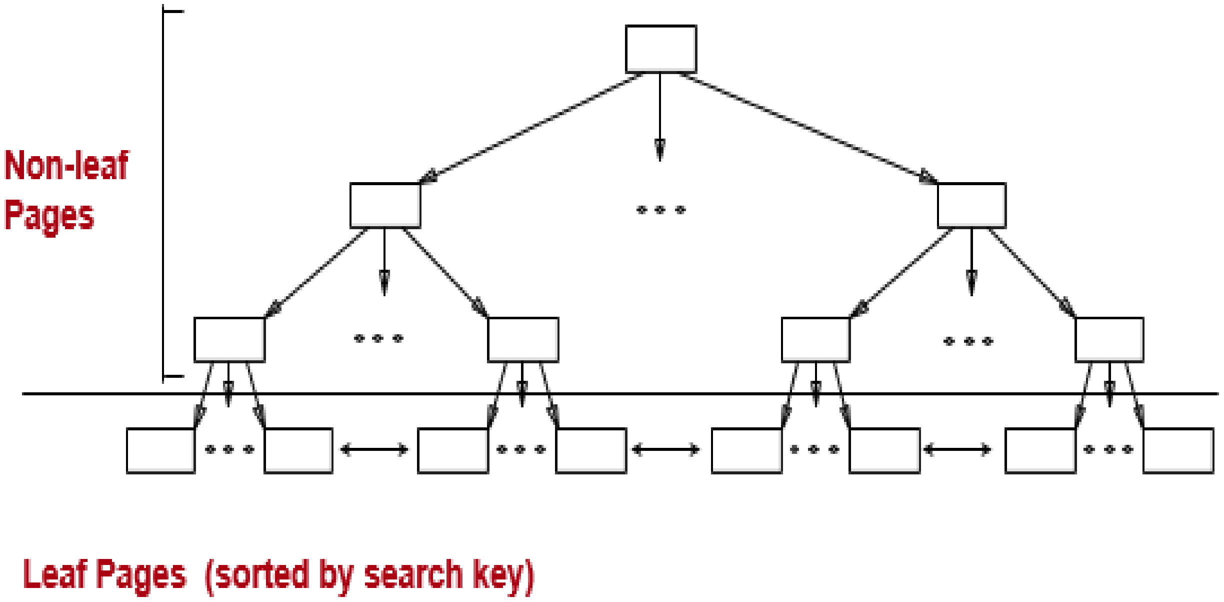
- Search trees
  - B does not mean binary!

- Idea in B Trees:
  - make 1 node = 1 physical page
  - Balanced, height adjusted tree (not the B either)

- Idea in B+ Trees:
  - Make leaves into a linked list (for range queries)

B+ Tree

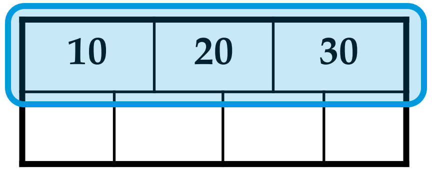
B+ Tree

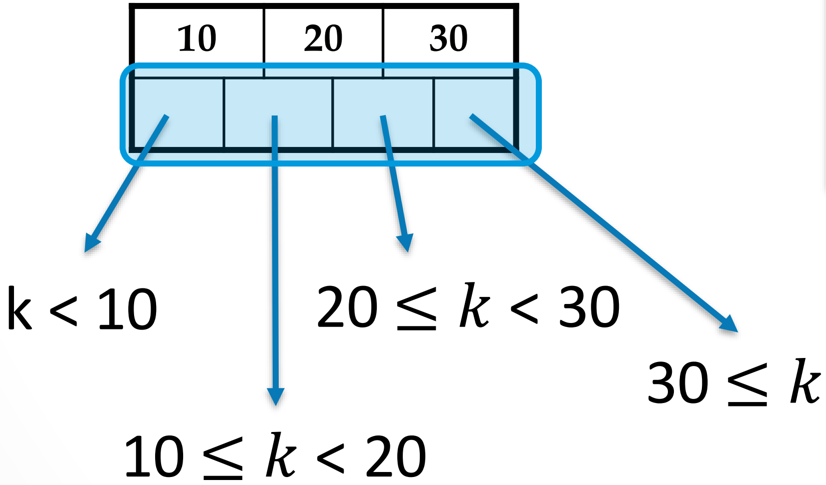


- Leaf pages contain data entries, and are chained (prev & next)

B+ Tree

B+ Tree basics





Parameter ***d*** = the order

Each *non-leaf* ***node*** has d $\leq m \leq$ 2d ***entries***

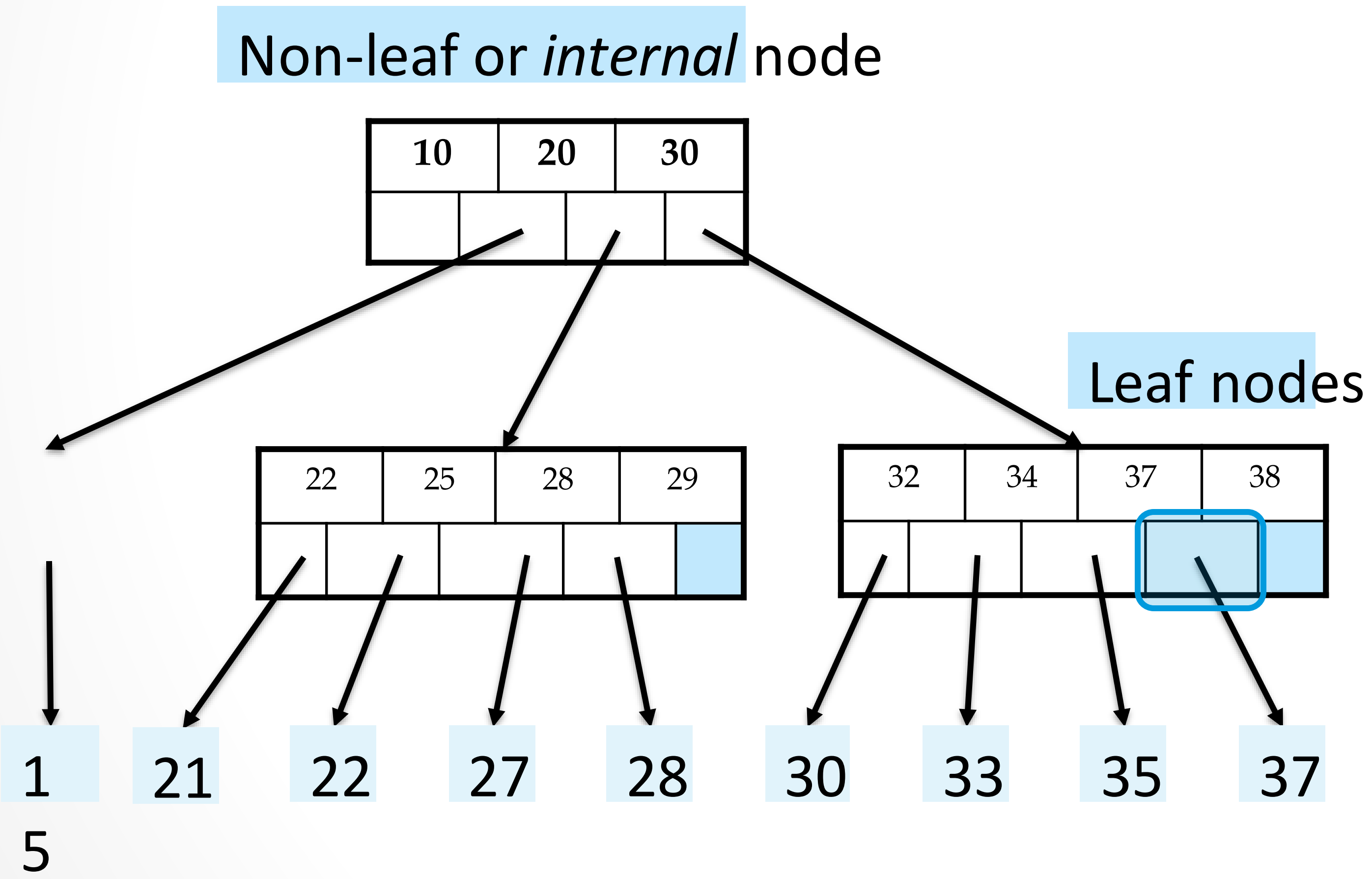
- ***Minimum 50% occupancy***

Root ***node*** has $1 \leq m \leq$ 2d ***entries***

The *n* entries in a node define *n+1* ranges

B+ Tree

B+ Tree basics

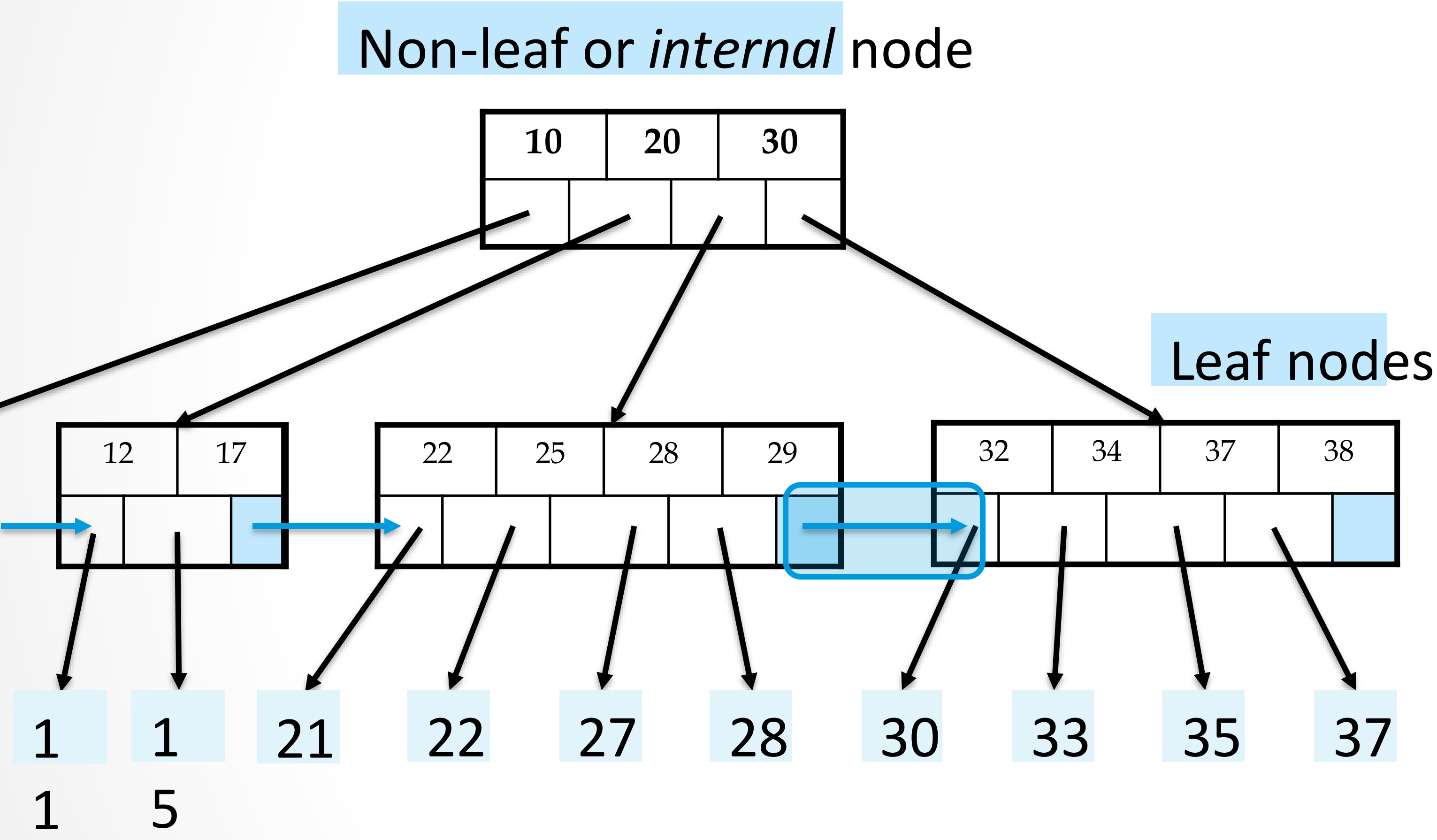


Leaf nodes also have between *d* and *2d* entries, and are different in that:

Their entry slots contain pointers to data records

B+ Tree

B+ Tree Basics



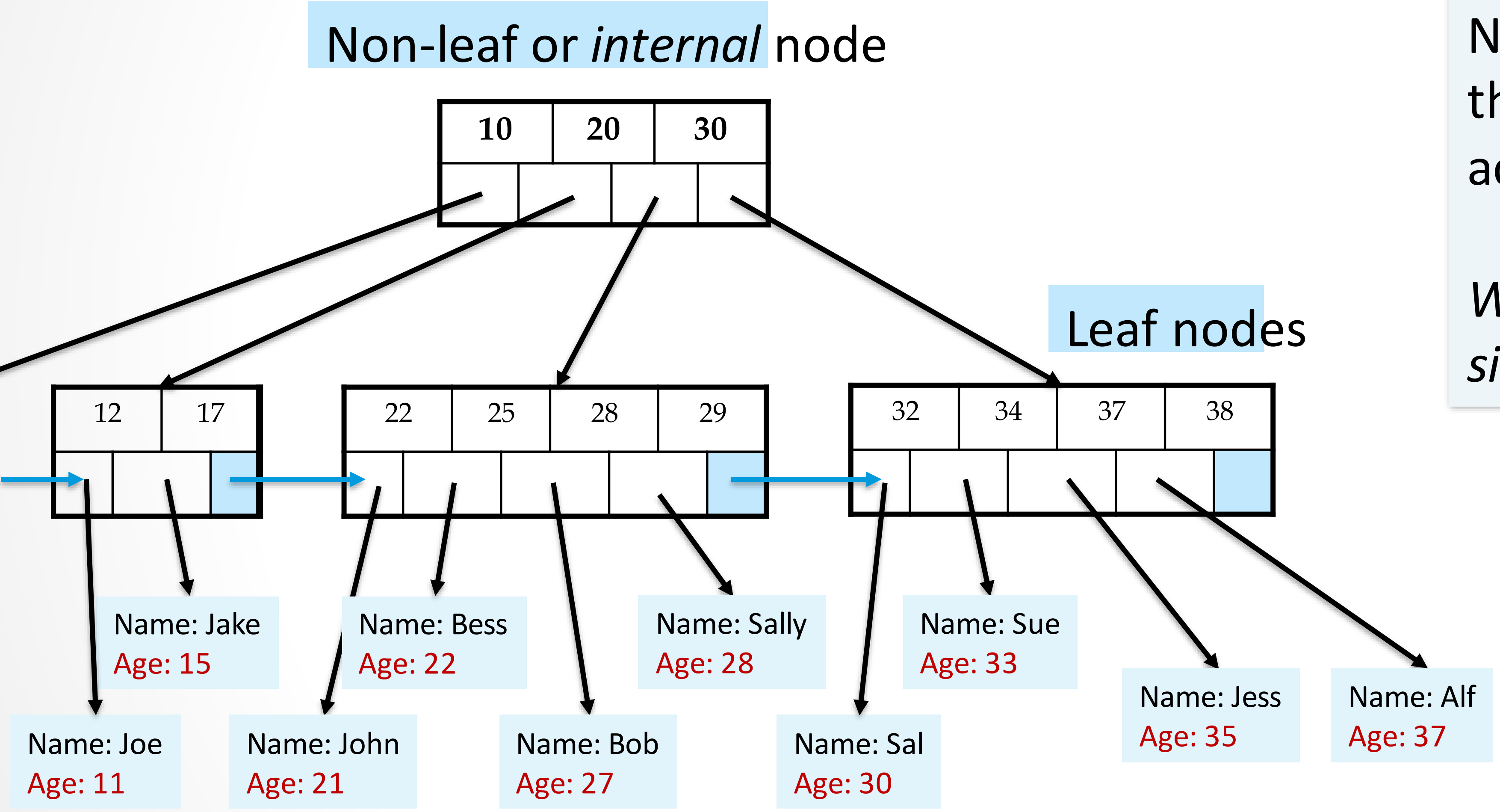
Leaf nodes also have between *d* and *2d* entries, and are different in that:

Their entry slots contain pointers to data records

They contain a pointer to the next leaf node as well, ***for faster sequential traversal***

B+ Tree

B+ Tree Basics

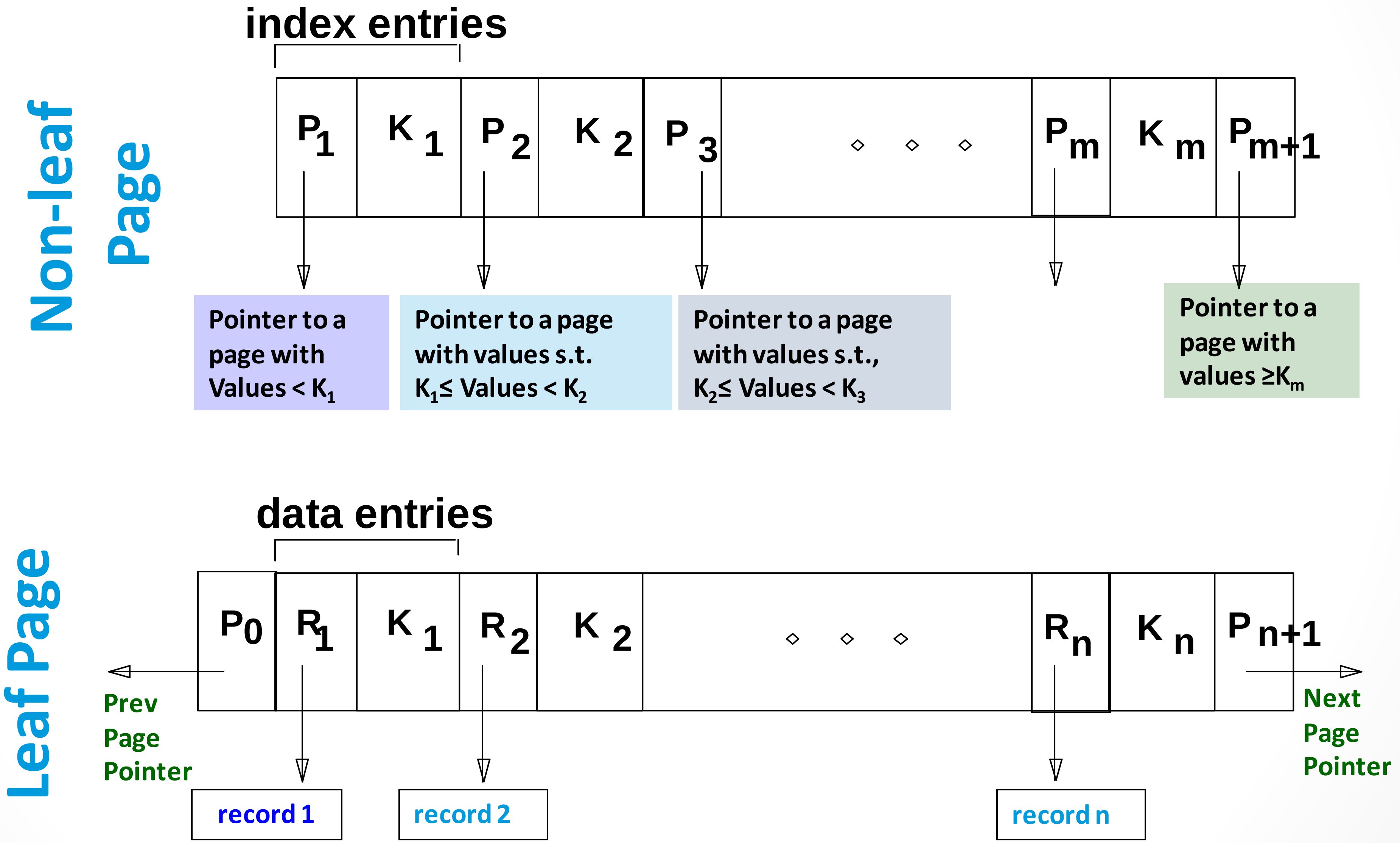


Note that the pointers at the leaf level will be to the actual data records (rows).

We might truncate these for simpler display (as before)…

B+ Tree

B+ Tree Page Format



B+ Tree

B+ Trees: Operations, Design & Cost

B+ tree supports the following operations:

- equality search
- range search
- insert
- delete

B+ Tree

Searching a B+ Tree

- For exact key values:
  - Start at the root
  - Proceed down, to the leaf

```
SELECT name
FROM   people
WHERE  age = 25
```

- For range queries:
  - As above
  - *Then sequential traversal*

```
SELECT name
FROM   people
WHERE  20 <= age
  AND  age <= 30
```

B+ Tree

B+ Tree: Search

- start from root
- examine index entries in non-leaf nodes to find the correct child
- traverse down the tree until a leaf node is reached
- non-leaf nodes can be searched using a binary or a linear search

B+ Tree

B+ Tree Search

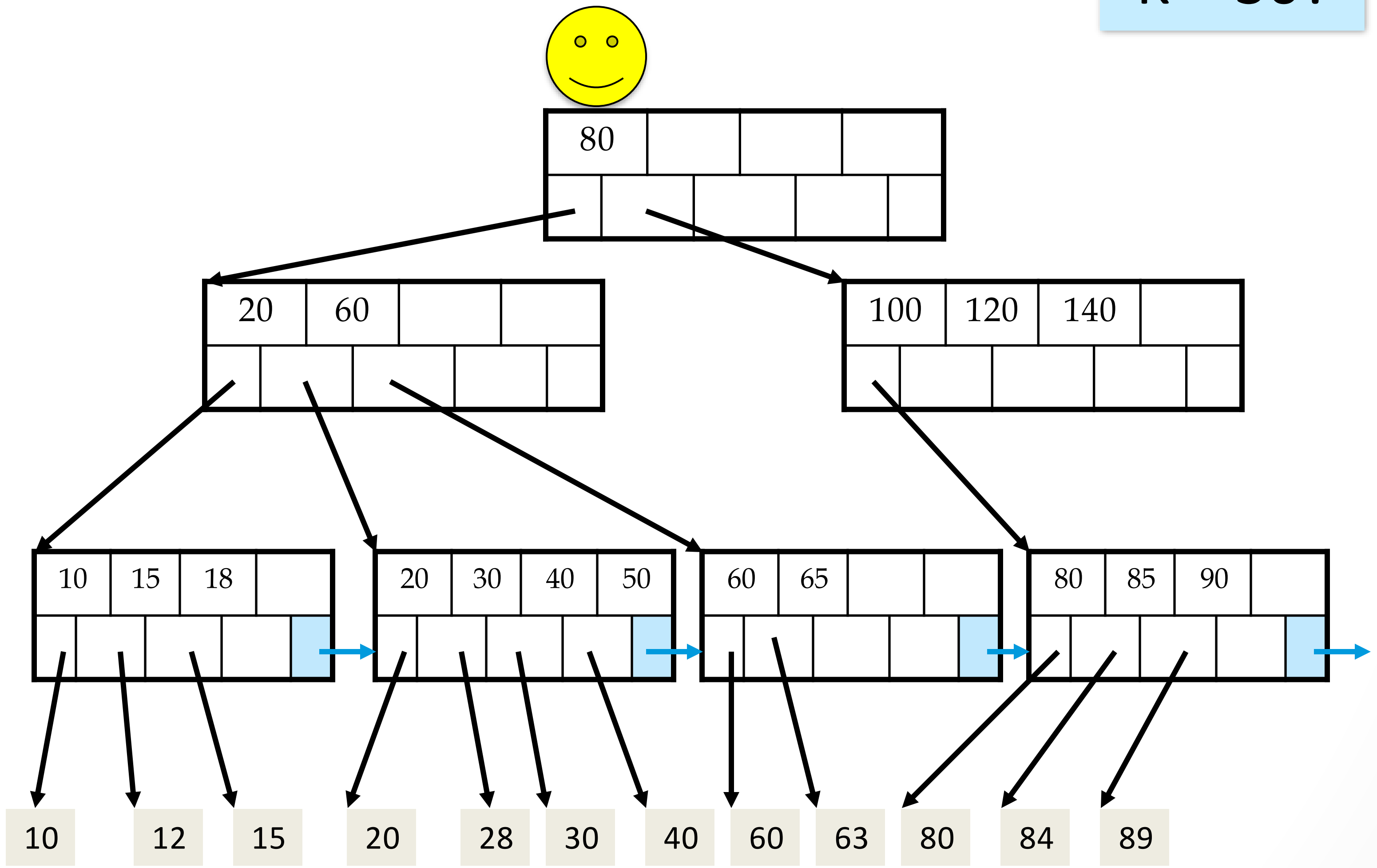
K = 30?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



Not all nodes pictured

B+ Tree

B+ Tree Range Search Animation

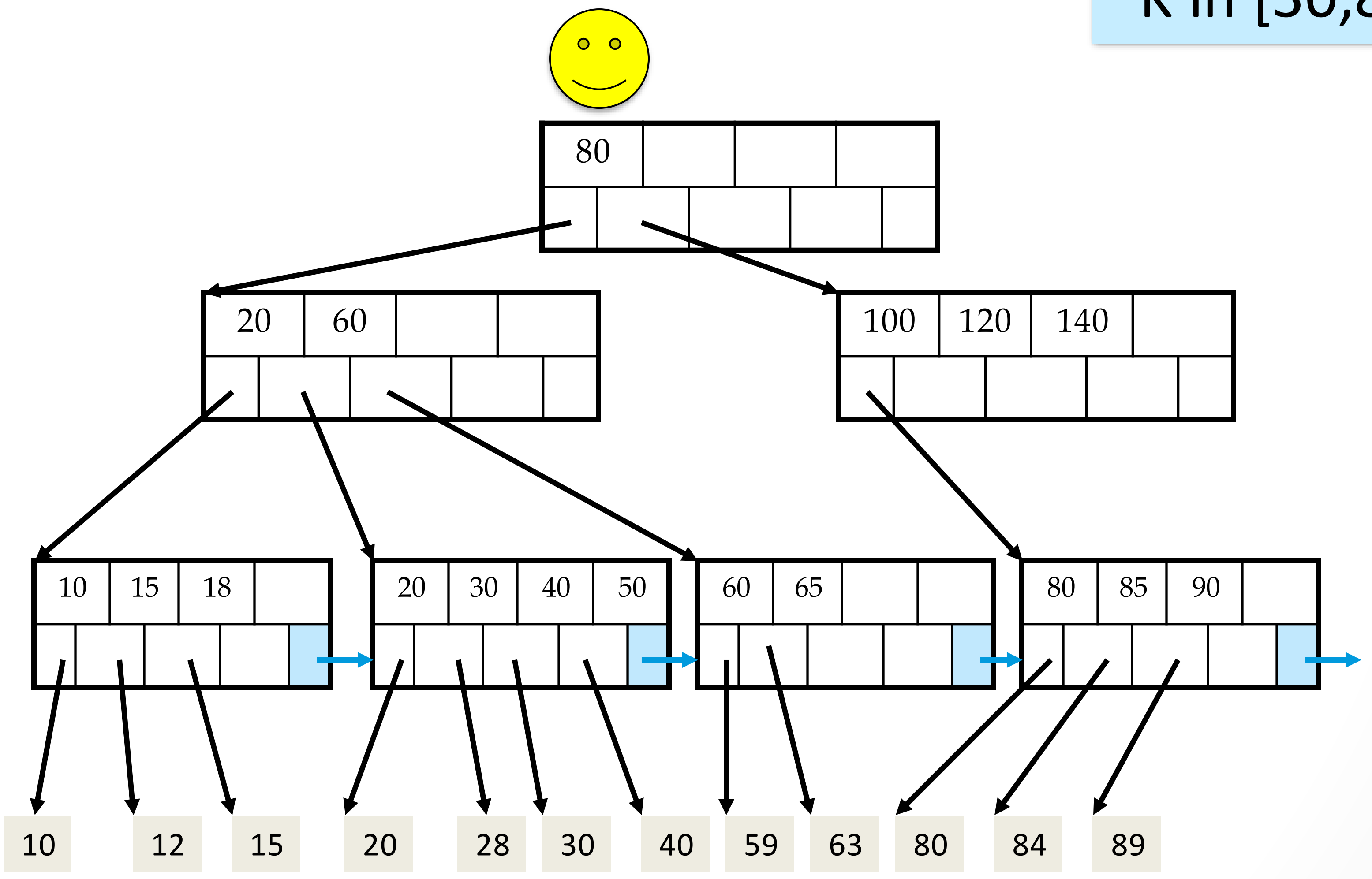
K in [30,85]?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



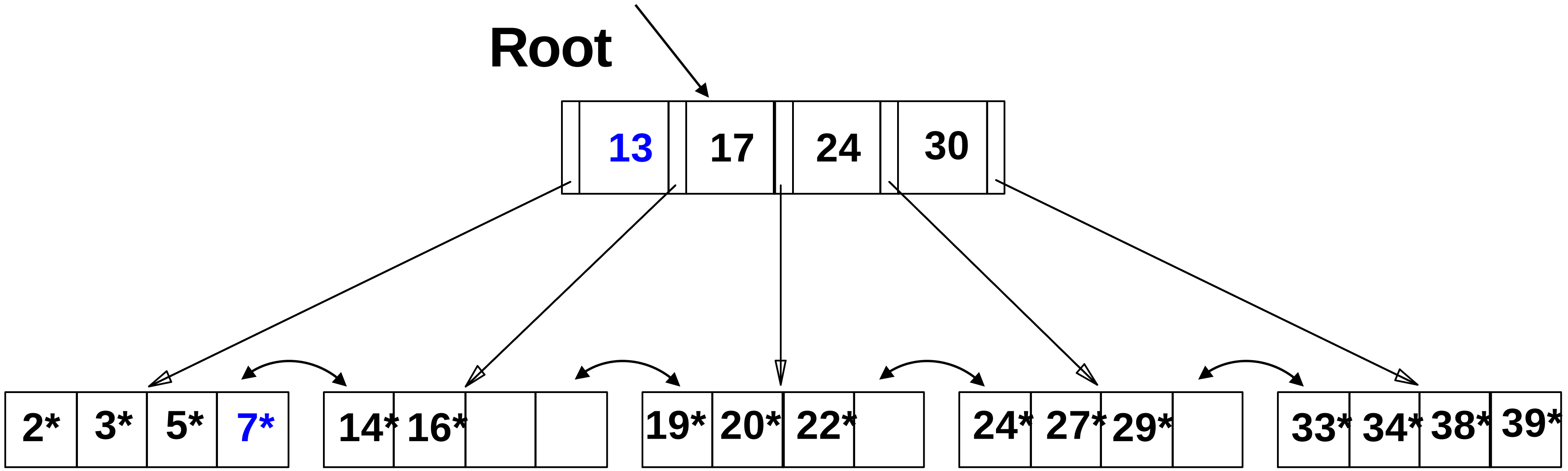
B+ Tree

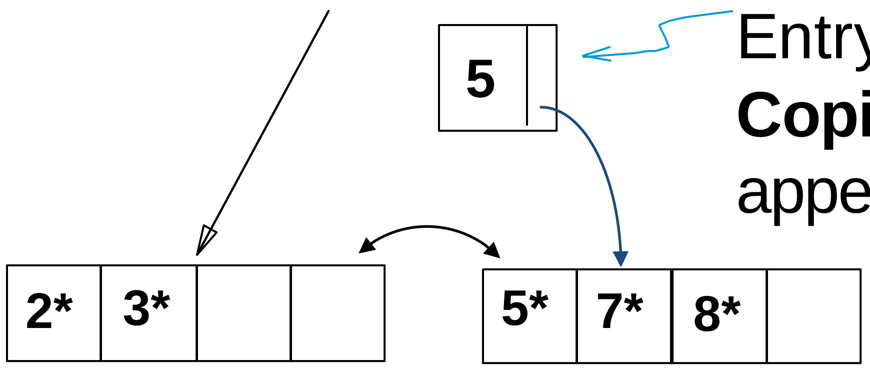
B+ Tree: Insert

- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must ***split*** *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split non-leaf node, redistribute entries evenly, but **pushing up** the middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

B+ Tree

Inserting 8* into B+ Tree



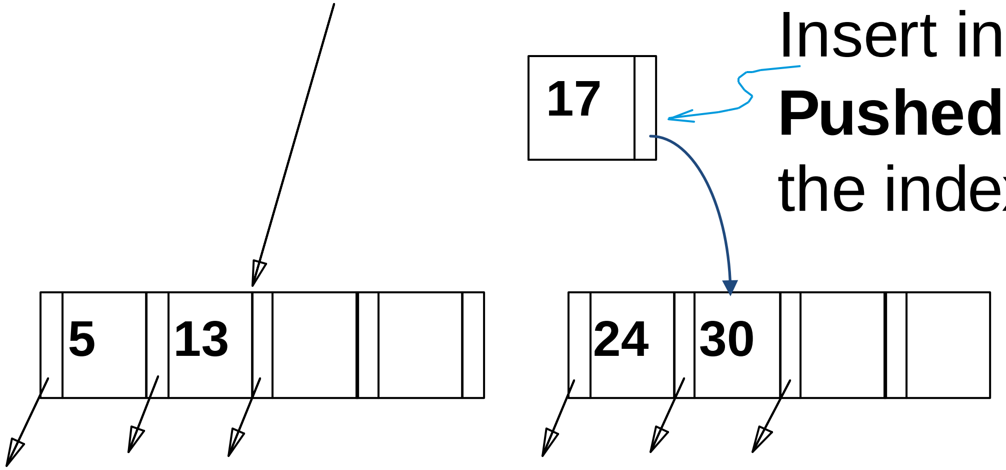


Entry to be inserted in parent node

Copied up (and continues to appear in the leaf)

B+ Tree

Inserting 8* into B+ Tree

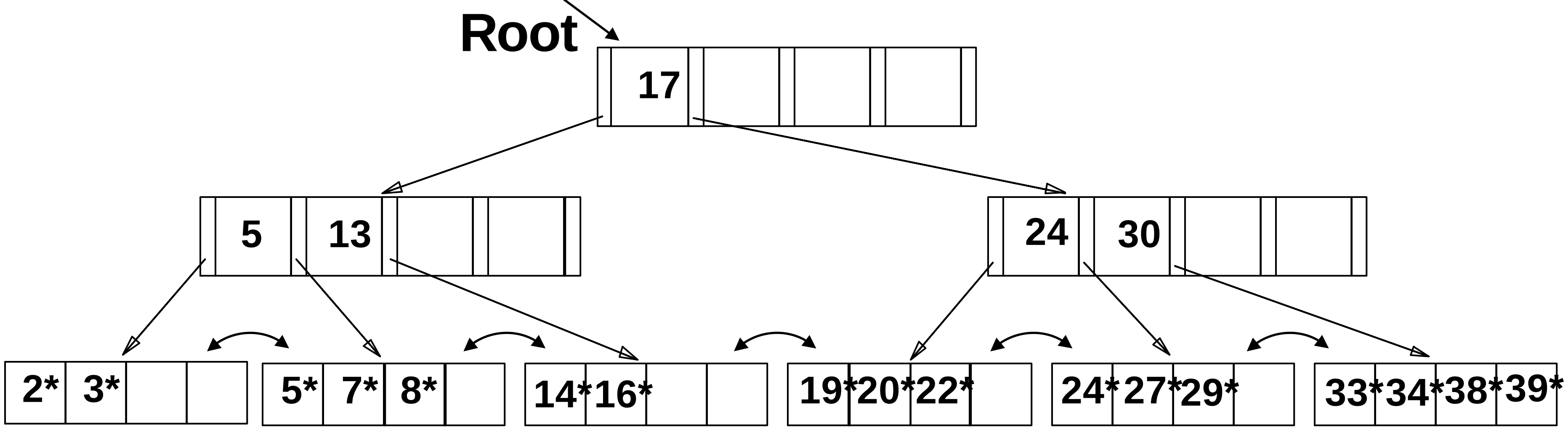


Insert in parent node.
Pushed up (and only appears once in the index)

Minimum occupancy is guaranteed in both leaf and index page splits

B+ Tree

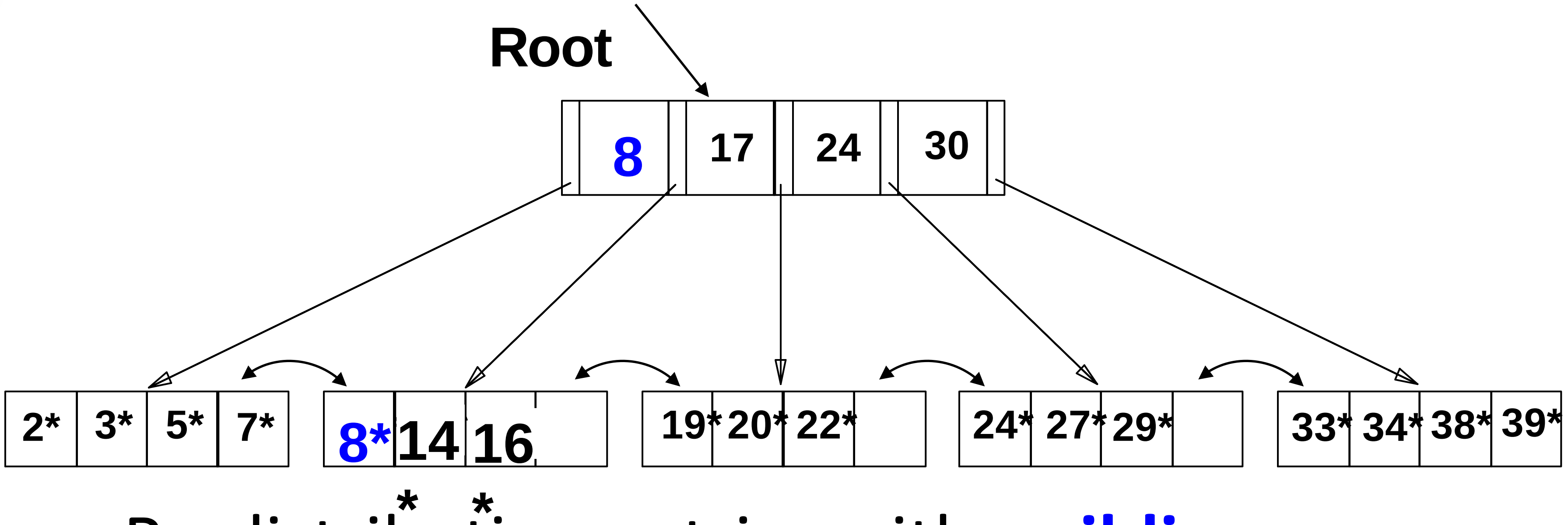
Inserting 8* into B+ Tree



- Root was split: height increases by 1
- Could avoid split by re-distributing entries with a sibling
  – Sibling: immediately to left or right, and same parent

B+ Tree

Inserting 8* into B+ Tree



- Re-distributing entries with a **sibling**
  - Improves page occupancy
  - Usually not used for non-leaf node splits. Why?
    - Increases I/O, especially if we check both siblings
    - Better if split propagates up the tree (rare)
    - Use only for leaf level entries as we have to set pointers

B+ Tree

Fast Insertions & Self-Balancing

- The B+ Tree insertion algorithm has several attractive qualities:
  - **~ Same cost as exact search**
  - ***Self-balancing:*** B+ Tree remains **balanced** (with respect to height) even after insert

B+ Trees also (relatively) fast for single insertions!
However, can become bottleneck if many insertions (if fill-factor slack is used up…)

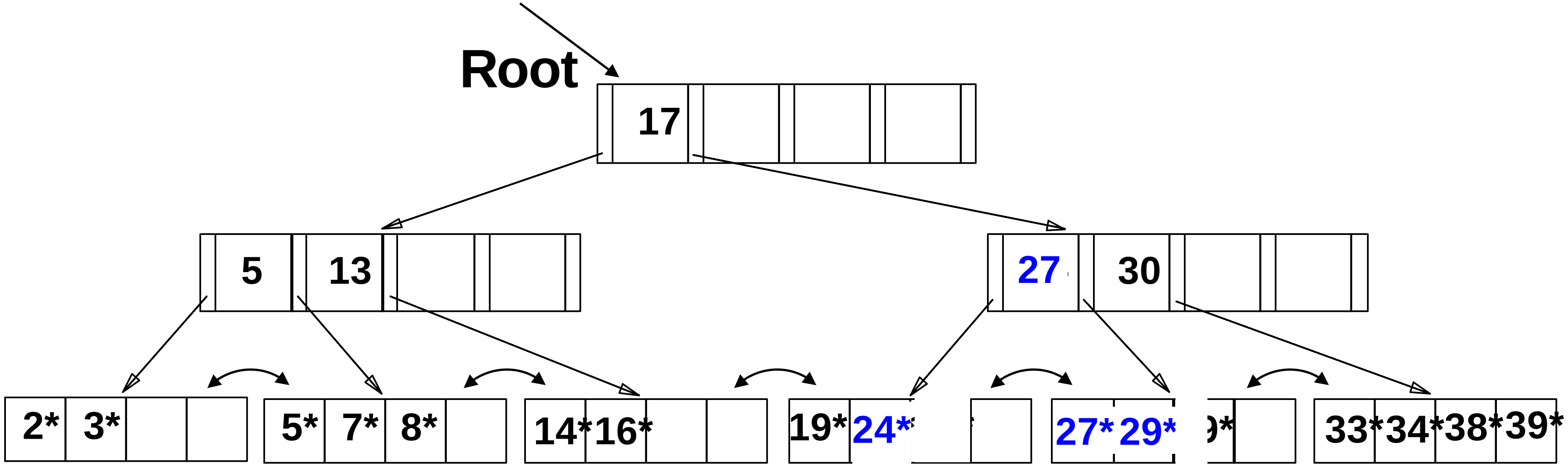
B+ Tree

B+ Tree: Deleting a data entry

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to **re-distribute**, borrowing from *sibling* *(adjacent node with same parent as L)*.
    - If re-distribution fails, ***merge*** *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could **propagate** to root, decreasing height.

B+ Tree

Deleting 22* and 20*



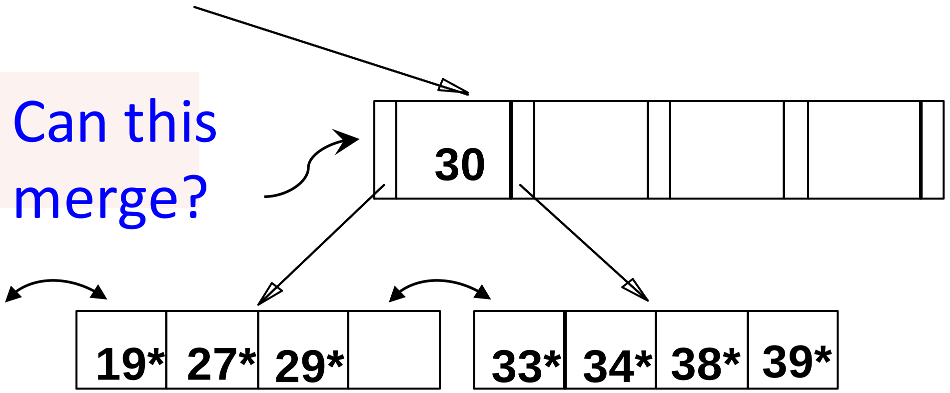
- Deleting 22* is easy.
- Deleting 20* is done with re-distribution. Notice how the middle key is **copied up**.

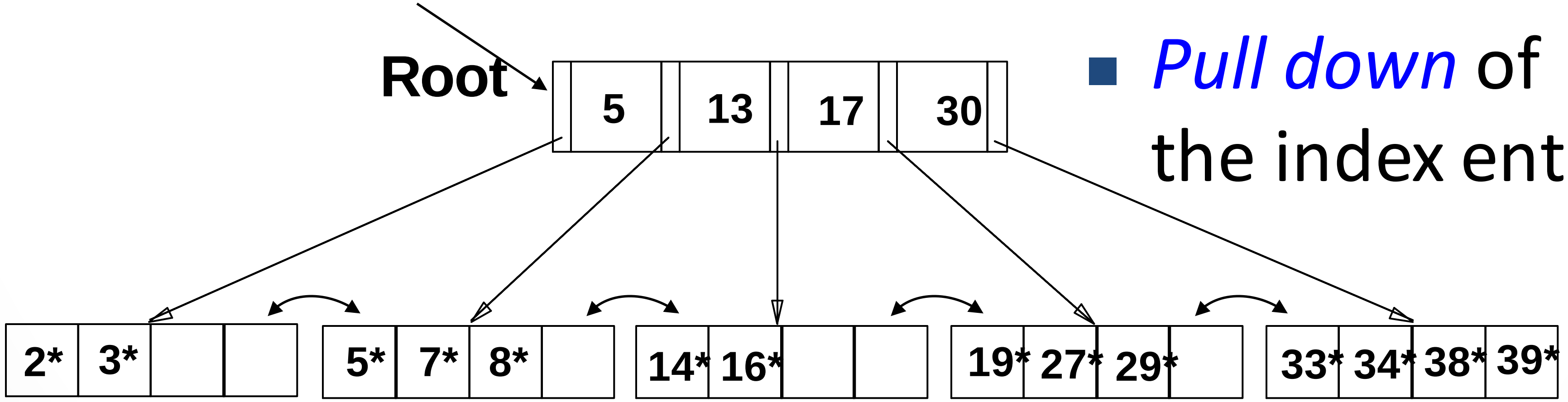
B+ Tree

And then deleting 24*

- Must merge.
- In the non-leaf node, ***toss*** the index entry with key value = 27

Can this merge?



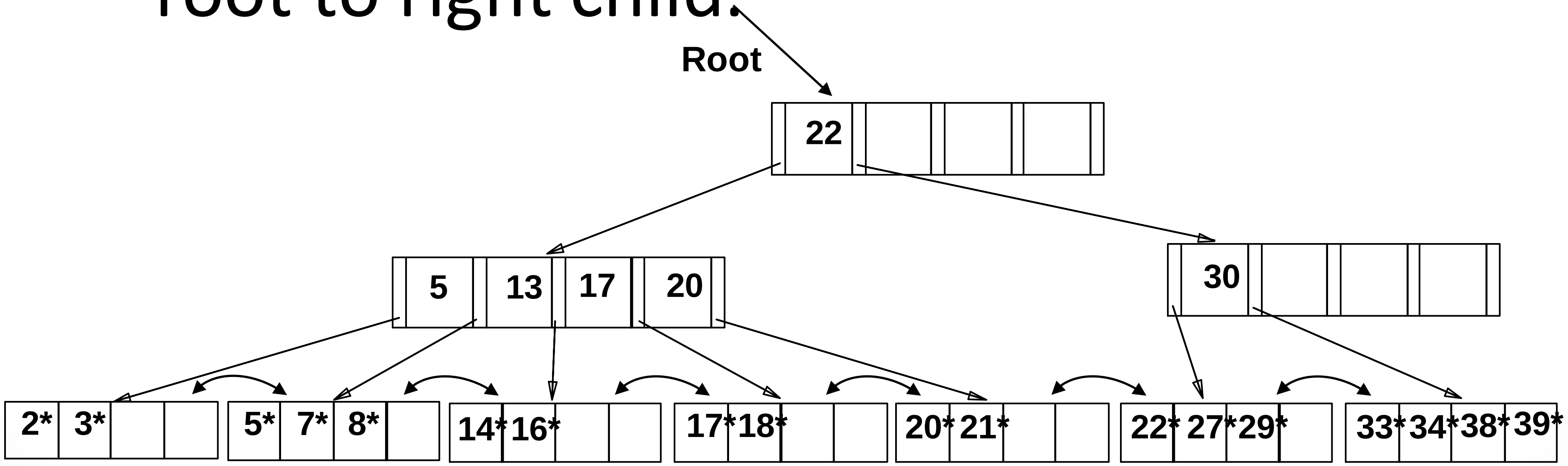


- *Pull down* of the index entry

B+ Tree

Non-leaf Re-distribution

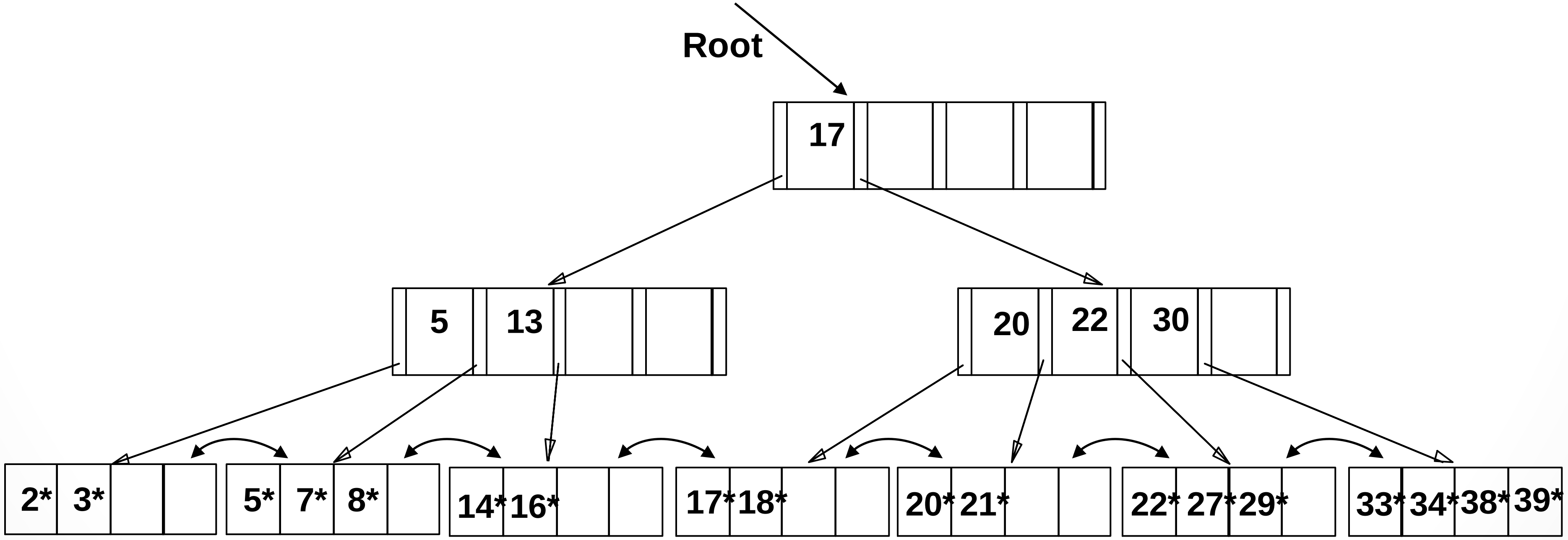
- Tree *during deletion* of 24*.
- Can re-distribute entry from left child of root to right child.



B+ Tree

After Re-distribution

- Rotate through the parent node
- It suffices to re-distribute index entry with key 20; For illustration 17 also re-distributed



B+ Tree

B+ Tree deletion

- Try redistribution with **all** siblings first, then merge. Why?
  - Good chance that redistribution is possible (large fanout!)
  - Only need to propagate changes to parent node
  - Files typically grow not shrink!

B+ Tree

B+ Tree Design

- How large is ***d***?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes

NB: Oracle allows 64K = 2^16 byte blocks
→ d <= 2730

- We want each *node* to fit on a single *block/page*
  - $2d \times 4 + (2d+1) \times 8 \le 4096 \rightarrow \boldsymbol{d \le 170}$

B+ Tree

B+ Tree: High Fanout = Smaller & Lower IO

- As compared to e.g. binary search trees, B+ Trees have **high *fanout*** (***between d+1 and 2d+1***)

- This means that the **depth of the tree is small** → getting to any element requires very few IO operations!
  - Also can often store most or all of the B+ Tree in main memory!

- A TiB = 2^{40} Bytes. What is the height of a B+ Tree (with fill-factor = 1) that indexes it (with 64K pages)?
  - $(2*2730 + 1)^h = 2^{40}$ → ***h = 4***

The **fanout** is defined as the number of pointers to child nodes coming out of a node

Note that fanout is dynamic- we'll often assume it's constant!

The known universe contains ~10^{80} particles… what is the height of a B+ Tree that indexes these?

B+ Tree

B+ Trees in Practice

- Typical order: d=100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Top levels of tree sit *in the buffer pool*:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

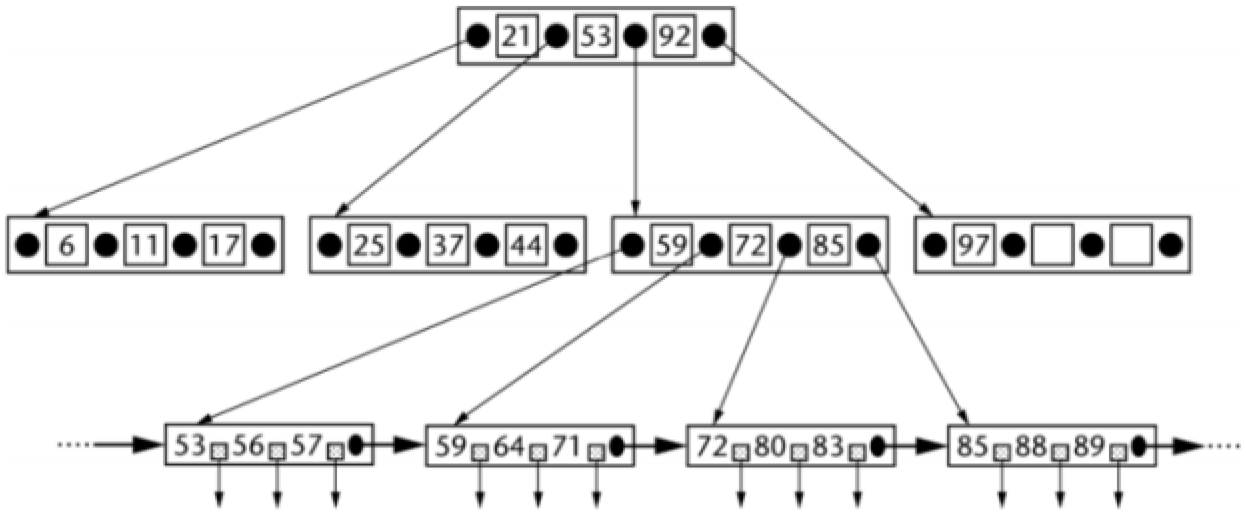
<u>Fill-factor</u> is the percent of available slots in the B+ Tree that are filled; is usually < 1 to leave slack for (quicker) insertions

Typically, only pay for one IO!

B+ Tree

Cost: Number of Block accesses : ***Read***

- B+ tree with **h** height : $h > \log n / \log p$

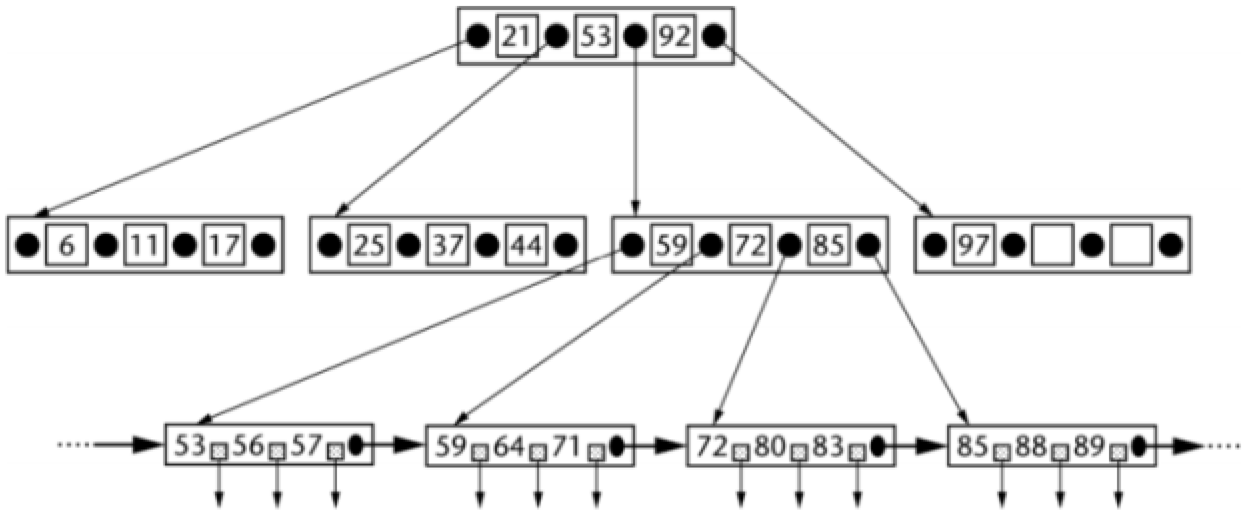


- Read a single row in a table (using a B+tree)
  = $h + 1$ block accesses.

B+ Tree

Cost: Number of Block accesses : ***Update***

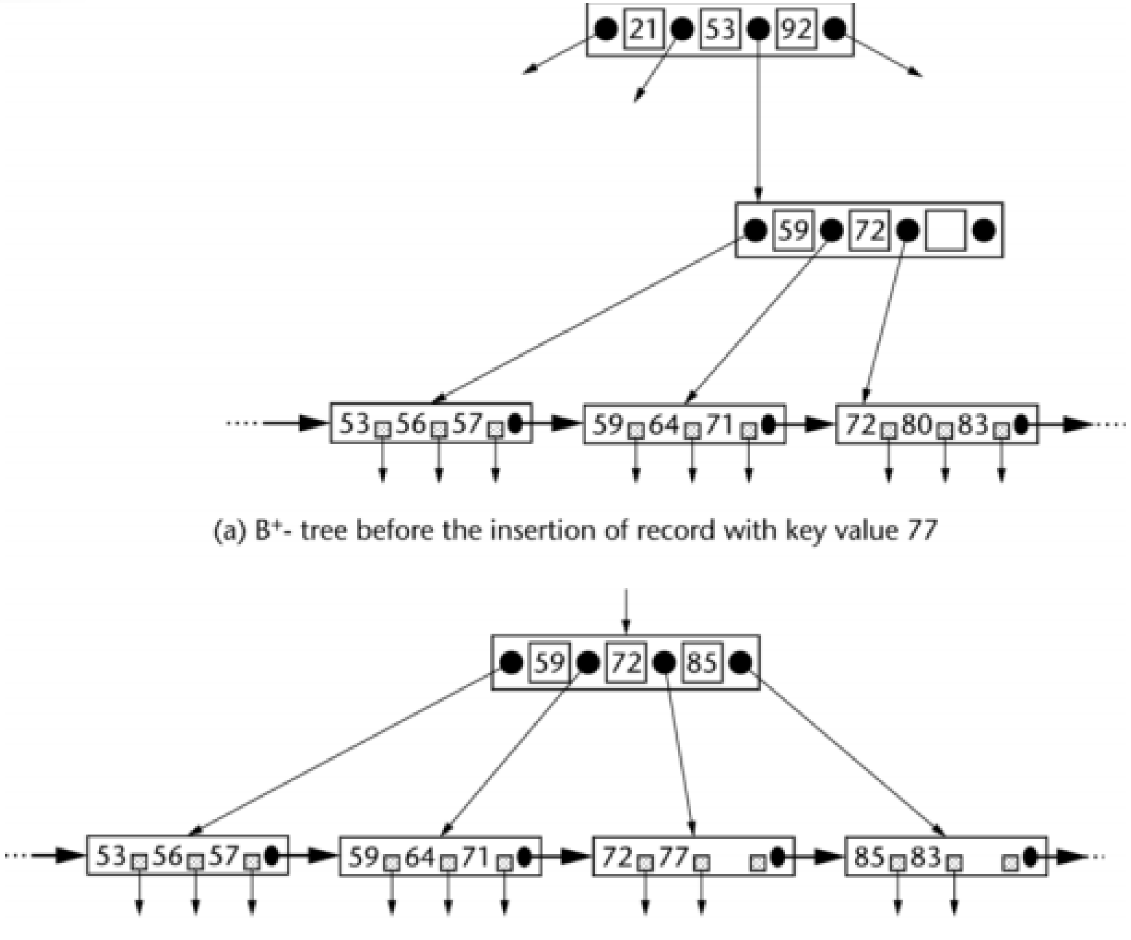
- B+ tree with **h** height



- Update cost for a single row (B+tree) = search cost + rewrite data block
  = (h + 1) + 1
  = h + 2 block accesses

B+ Tree

Cost: Number of Block accesses : ***Insert***



(a) B+- tree before the insertion of record with key value 77

(b) B+- tree after the insertion and split block operation

Insert cost for a single row (B+tree)
= search cost + rewrite data block + rewrite index block
= (h + 1) + 1 + 1
= h + 3 block accesses,

B+ Tree

Cost: Number of Block accesses : ***Delete***

Deletions may result in emptying a data block or index node, which necessitates the consolidation of two nodes into one.
This may require a rewrite of the leaf index node to reset its pointers. The empty data node can be either left alone or rewritten with nulls, depending on the implementation

Delete cost for a single row (B+tree)
= search cost + rewrite data block + rewrite index block
= $(h + 1) + 1 + 1$
= $h + 3$ block accesses

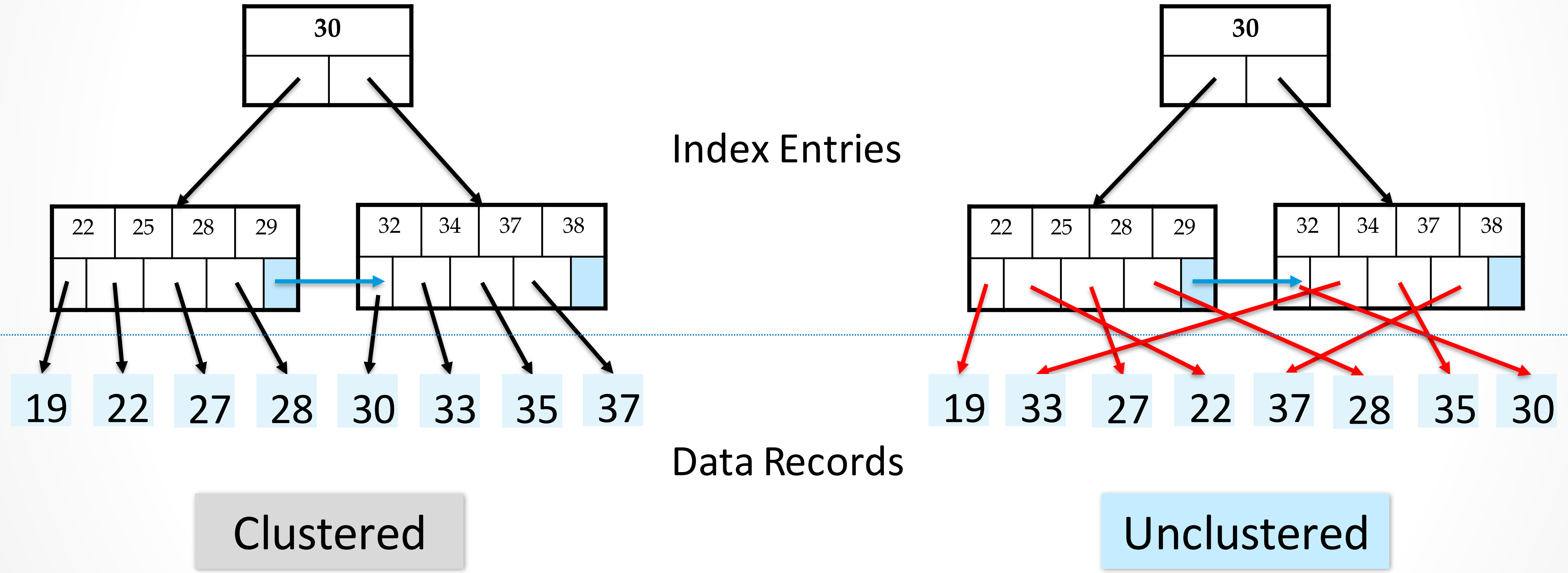
B+ Tree

Clustered Indexes

An index is ***<u>clustered</u>*** if the underlying data is ordered in the same way as the index's data entries.

B+ Tree

Clustered vs. Unclustered Index



B+ Tree

Clustered vs. Unclustered Index

- Recall that for a disk with block access, **sequential IO is much faster than random IO**
- For exact search, no difference between clustered / unclustered
- For range search over R values: difference between **1 random IO + R sequential IO**, and **R random IO**:
  - A random IO costs ~ 10ms (sequential much much faster)
  - For R = 100,000 records- **difference between ~10ms and ~17min!**

B+ Tree

Summary

- We create **indexes** over tables in order to support ***fast (exact and range) search*** and ***insertion*** over ***multiple search keys***

- **B+ Trees** are one index data structure which support very fast exact and range search & insertion via ***high fanout***
  - ***Clustered vs. unclustered*** makes a big difference for range queries too

B+ Tree

B^+-tree indices are an alternative to indexed sequential files

- Advantage of B^+-tree index files: automatically reorganizes itself with small, local, changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance.
- Advantages of B^+-trees outweigh disadvantages, and they are used extensively
- Disadvantage of indexed-sequential files: performance degrades as file grows, since many overflow blocks get created. Periodic reorganization of entire file is required.
- Disadvantage of B^+-trees: extra insertion and deletion overhead, space overhead.

Composite Index Search

Composite Index

- Like a **single index**, a **composite index** is also a data structure of records sorted on something. But unlike a **single index**, that something is not a **field**, but a concatenation of **multiple fields**.
- **Composite Index:** define indices on multiple columns. This index is called a Multi-column / Composite / Compound index.

```
SELECT empNo, empName, empAddress, jobTitle
    FROM employee
    WHERE jobTitle = 'database administrator'
    AND city = 'Los Angeles'
    AND totalPur < 500;
```

Composite Index Search

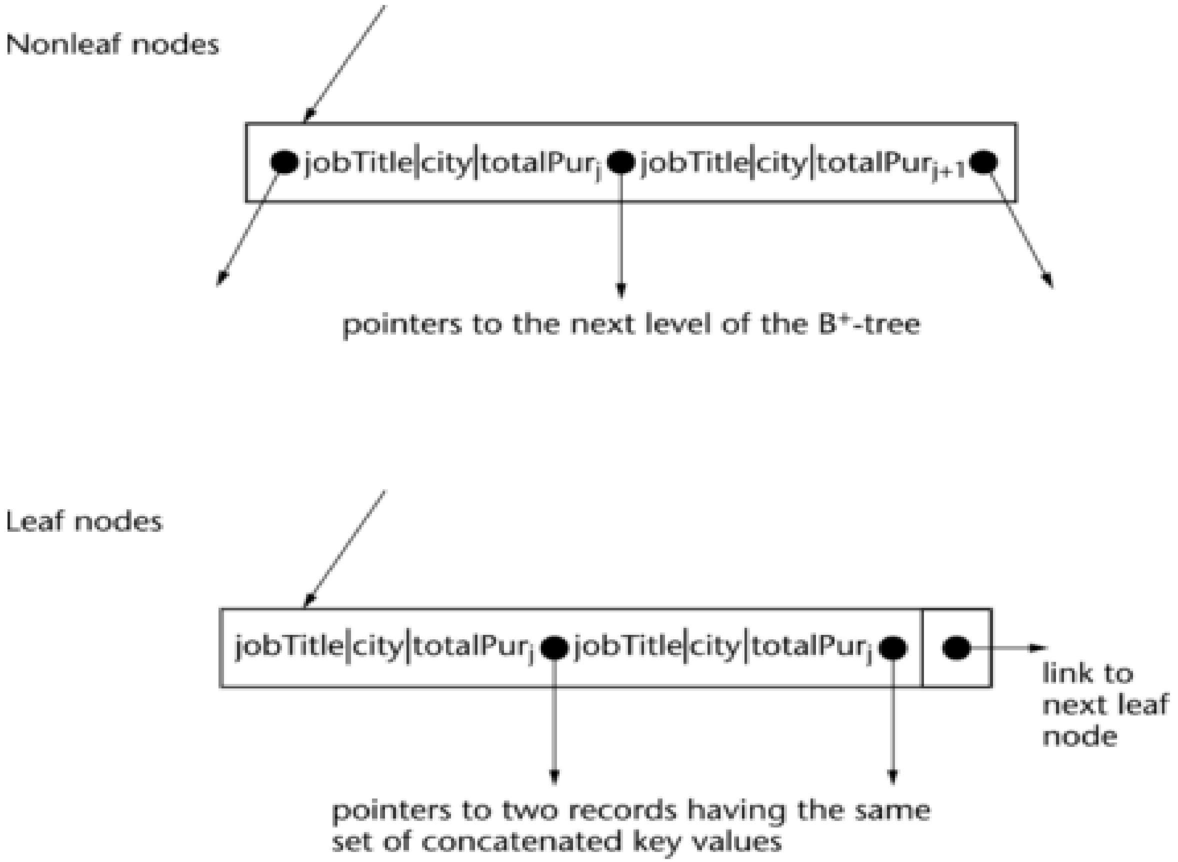
How does composite index work?

- The columns used in composite indices are concatenated together, and those concatenated keys are stored in sorted order using a B+ Tree.
- When you perform a search, concatenation of your search keys is matched against those of the composite index
- In a nonunique index, the key field is a concatenation of all the attributes you want to set up to access the set of rows desired
- Example:

  create index NameIndex
  on Table1(A1,A2,A3) using B+TREE

Composite Index Search

How does composite index work?



B+tree search of concatenated keys is much faster than searching for the target rows over and over for each individual key using multiple simple indexes

Example

Customer rows have attributes for customer name, customer number,
street address, city, state, zip code, phone number, e-mail, employer, job title, credit rating, date of last purchase, and total amount of purchases

- A table of 10 million rows
- Row size is 250 bytes; block size is 5,000 bytes; pointer size, including row offset, is 5 bytes; and composite key for jobTitle, city, and totalPur is 35 bytes. The table of 10 million rows has 500,000 blocks since each block contains 20 rows

SQL.

```
SELECT empNo, empName, empAddress, empPhone, empEmail
FROM customer
WHERE jobTitle = 'software engineer'
AND city = 'Chicago'
AND totalPur > 1000
```

For each AND condition we have the following hit rates

Job title (jobTitle) is ‘software engineer’: 84,000 rows.
City (city) is ‘Chicago’: 210,000 rows.
Total amount of purchases (totalPur) > $1,000: 350,000 rows.
Total number of target rows that satisfy all three conditions = 750.
Query cost ??

Query cost

- If we assume the total of 84,000 + 210,000 + 350,000 = 644,000 pointers fit into blocks holding (5,000 bytes/35 bytes = 142) pointers each
- cost of using the three index approach is :
  = three index search cost + merge pointer cost
  + final data access cost
  = (h1 + 1) + (h2 + 1) + (h3 + 1) + 644,000/142
  + 750 block accesses = 3 + 4 + 4 + 4,535 + 750 block accesses.
- Block access time
  = 2 ms rotational delay + transfer time of 5,000 bytes/320 MB/sec
  = 2.02 ms.
  Query I/O time
  = 761 × 2.02 ms + (4,535 blocks × 5,000 bytes/block)/320 MB/sec

  = 1.54 sec + .07 sec
  = 1.61 seconds.

Query cost **Composite Index Approach**

- 5,000 bytes/block >= $p \times 5$ for pointers + $(p - 1) \times 35$ for index entries => $p = 125$
- h > log n / log p = log 10,000,000/log 125 = 7/2.097 = 3.34, or $h = 4$
- Number of blocks needed to hold 750 target row pointers
  = 750/(5,000 bytes/block/35 bytes/pointer) = 750/142
  = 5.28 blocks => 6 blocks.
- **Query cost**
  = composite index search + pointer search for target records
  + final data access cost
  = h + 6 + 750
  = 4 + 6 + 750 block accesses.
  Query I/O time
  = 754 × 2.02 ms + (6 blocks × 5,000 bytes/block)/320 MB/sec
  = 1.52 seconds.

Bitmap Indexing

Bitmap Indexing in DBMS

Bitmap Indexing is a special type of database indexing that uses bitmaps. This technique is used for huge databases, when column is of low cardinality and these columns are most frequently used in the query.

Lastname	Firstname	Country
Müller	Heinrich	GER
Miller	Henry	UK
Magritte	René	FRA
Smith	John	US
Starkey	Richard	UK
Weiser	Bud	US
Röder	Hasso	GER
Hugo	Victor	FRA
...	...	...

Bitmap
Index on Country

FRA	GER	UK	US
0	1	0	0
0	0	1	0
1	0	0	0
0	0	0	1
0	0	1	0
0	0	0	1
0	1	0	0
1	0	0	0
...	...	...	...

Bitmap Indexing

Storing the Bitmap index

- One bitmap for each value, and one for Nulls
- Need to store each bitmap
- Simple method: 1 file for each bitmap
- Can compress the bitmap!

Index size?   #tuples * (cardinality of the domain + 1)   *bits*

When is a bitmap index more space efficient than a B+-tree?

#distinct values < data entry size in the B+-tree

Bitmap Indexing

Advantages Disadvantages of Bitmap

northwest

farwest	1	0	0	1	0	0	1	0	0	1	0	0
southwest	0	0	0	0	1	1	0	0	1	0	0	1

SalesId bitmaps

410	0	1	0	0	0	1	0	0	0	1	0	0
411	0	0	0	1	0	0	0	1	0	0	1	0
412	1	0	0	0	1	0	1	0	1	0	0	0
415	0	0	1	0	0	0	0	0	0	0	0	1

Reg.name "southwest" AND salesId 412

southwest	0	0	0	0	1	1	0	0	1	0	0	1

AND

412	1	0	0	0	1	0	1	0	1	0	0	0

|| *Intersection bitmap*

RESULT	0	0	0	0	1	0	0	0	1	0	0	0

Advantages

Efficiency in terms of insertion deletion and updation

Faster retrieval of records

Disadvantages

Only suitable for large tables

Bitmap Indexing is time consuming

. . .

How to choose indexes

- Attributes in **`WHERE`** clause are candidates for index keys
  - exact match condition suggests hash index
  - indexes also speed up joins (later in class)
  - range query suggests tree index (B+ tree)

- Multi-attribute search keys should be considered when a **`WHERE`** clause contains several conditions
  - order of attributes is important for range queries
  - such indexes can enable index-only strategies for queries

. . .

How to choose indexes

Composite search keys: search on a combination of fields (e.g. <date, price>)

- equality query: every field value is equal to a constant value
  - date=“02-20-2015” and price =75
- range query: some field value is not a constant
  - date=“02-20-2015”
  - date=“02-20-2015” and price > 40

…

How to choose indexes

- Tips: don’t use indexes

  - on small relations,
  - on frequently modified attributes,
  - on non selective attributes (queries which returns ≥ 15% of data)
  - on attributes with values long string

Define indexes on primary and foreign keys

- Consider the definition of indexes on attributes used on queries which requires sorting: ORDER BY, GROUP BY, DISTINCT, Set operations

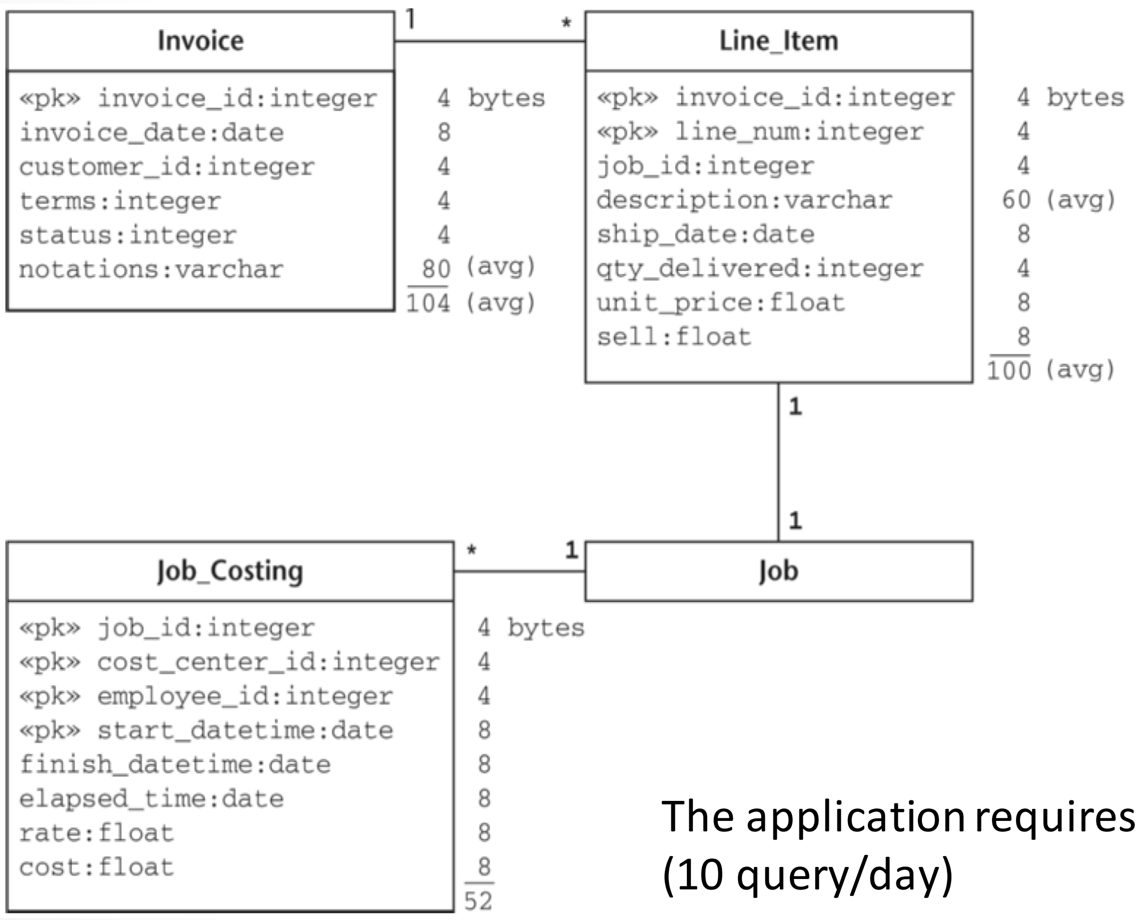
Selecting materialized view

- Simple View Materialization
- Exploiting commonality
- Exploiting Grouping and Generalization
- Resource Considerations
- Tips and Insights for Database Professionals

Simple View Materialization

- A materialized view (MV) is a database object that stores the result of a specific query.
- There are two ways that materialized views can be accessed.
  - The first is the brute force method where the SQL is written to explicitly access the view.
  - The second is for the decision to be made by the query compiler during query optimization.
- Effectively a materialized view caches calculations, permitting the reuse of the results, which leads to faster query responses and disk I/O.

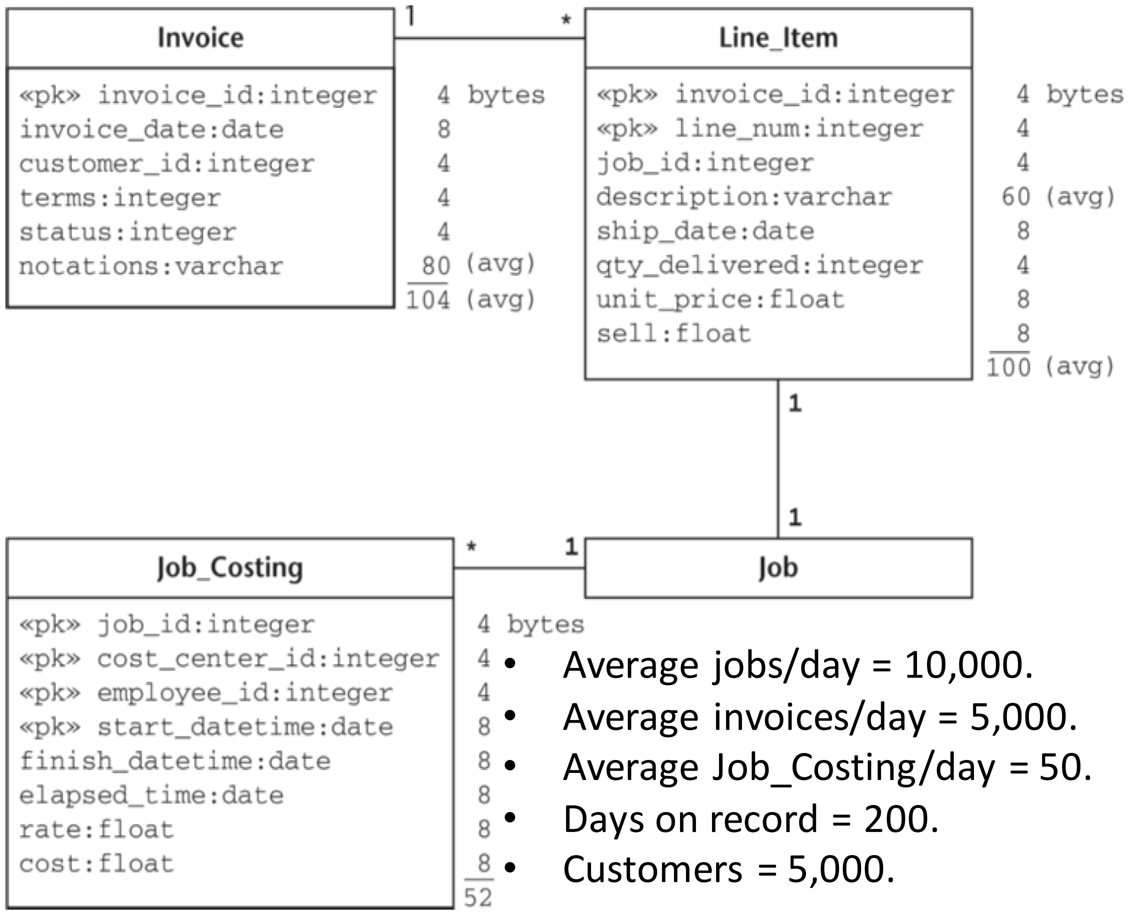
Simple View Materialization



- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.
- The row in all tables are clustered in primary key order.
- All tables can be sorted in memory.
- IBM U320 146 GB hard drive:
  - I/O time (4KB block in a dedicated disk) = 2.0 ms.
  - I/O time (64KB buffer in a dedicated disk) = 2.2 ms.
  - I/O time (4KB block in a shared disk) = 5.6 ms.
  - I/O time (64KB buffer in a shared disk) = 5.8 ms.

The application requires the cost, sell and profit for each job.
(10 query/day)

Simple View Materialization



- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.

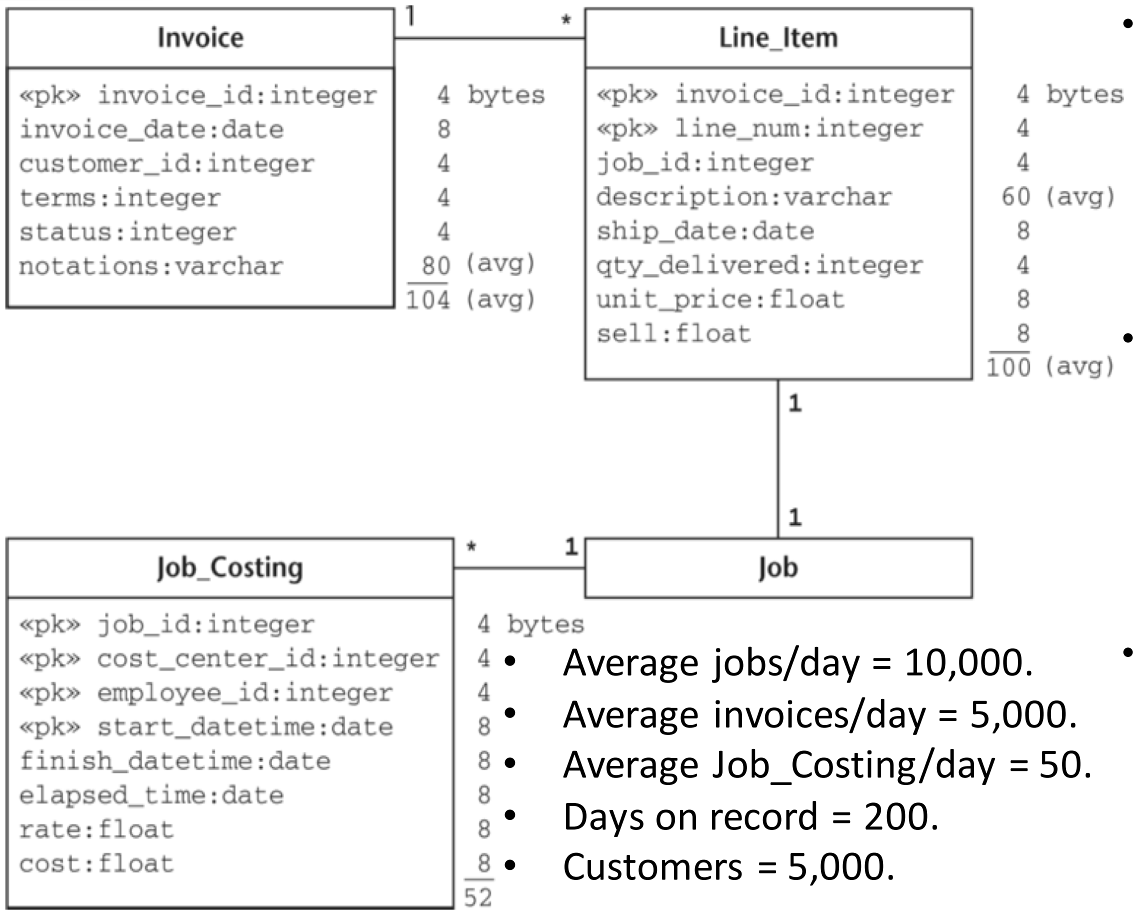
```
SELECT c.job_id,
  sum(c.cost) AS cost,
  sum(li.sell) AS sell,
  sum(li.sell) - sum(c.cost) AS profit
FROM Job_Costing AS c, Line_Item AS li
WHERE c.job_id = li.job_id
GROUP BY c.job_id;
```

- Query I/O time:

  Join cost (shared disk)

  = scan time (Line_Item and Job_Costing tables using 64 KB prefetch buffers)

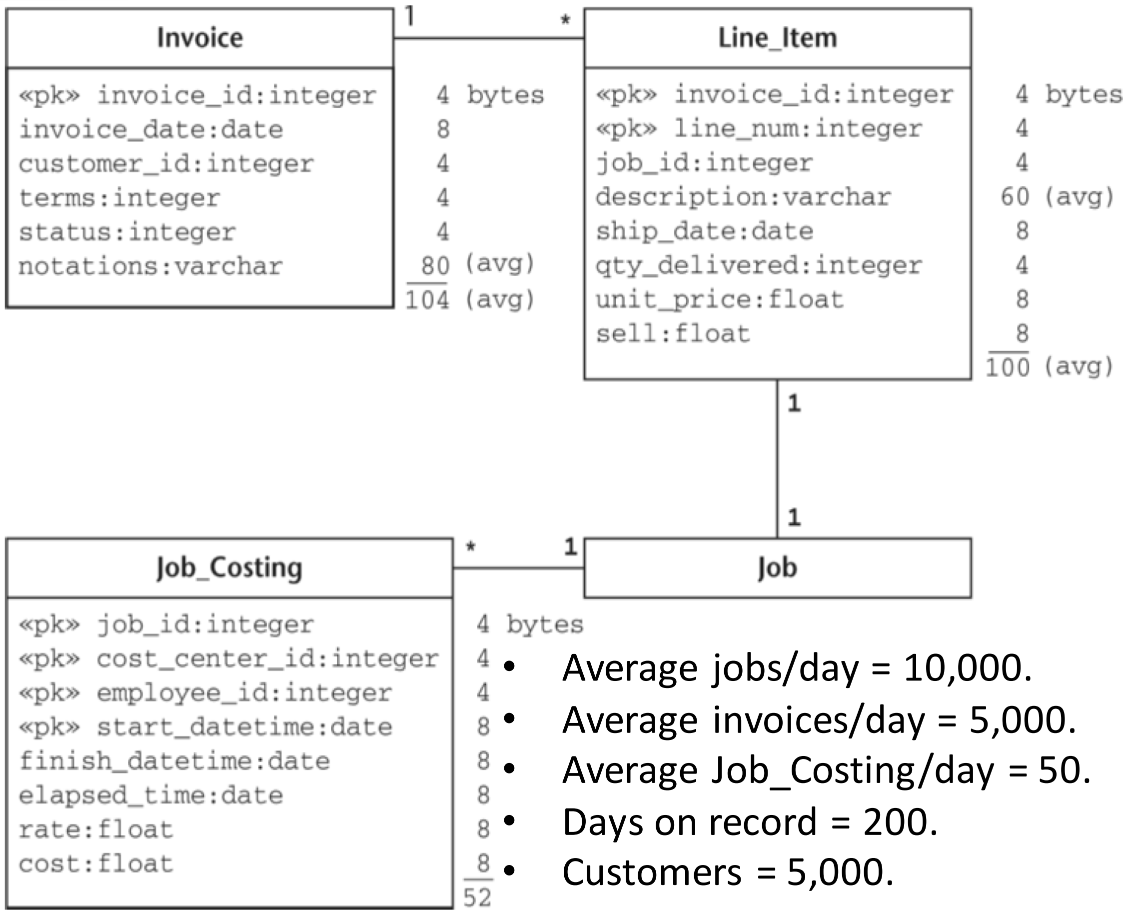
Simple View Materialization

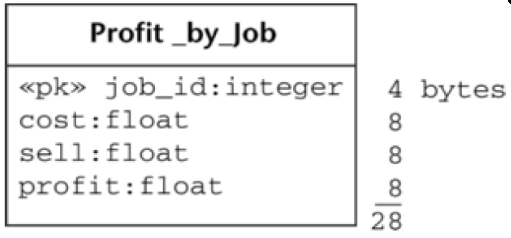


- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.

- The Line_Items calculations are:
  - Average rows per prefetch buffer
    - = floor((65,536 bytes/ buffer)/(100 bytes/row))
    - = 655.
  - Number of buffers
    - = ceiling((10,000*200) rows/(655 rows/buffer))
    - = 3,054.
- The Job_Costing calculations are:
  - Average rows per prefetch buffer
    - = floor((65,536 bytes/ buffer)/(52 bytes/row))
    - = 1,260.
  - Number of buffers
    - = ceiling((50*10,000*200) rows/(1,260 rows/buffer))
    - = 79,366.
- Query I/O time:
  - Join cost (shared disk)
    - = scan time (Line_Item and Job_Costing tables using 64 KB prefetch buffers)
    - = (3,054 + 79,366 buffers) * 5.8 ms
    - = 82,420 * 5.8 ms
    - = 478 seconds ≈ 8 minutes

Simple View Materialization



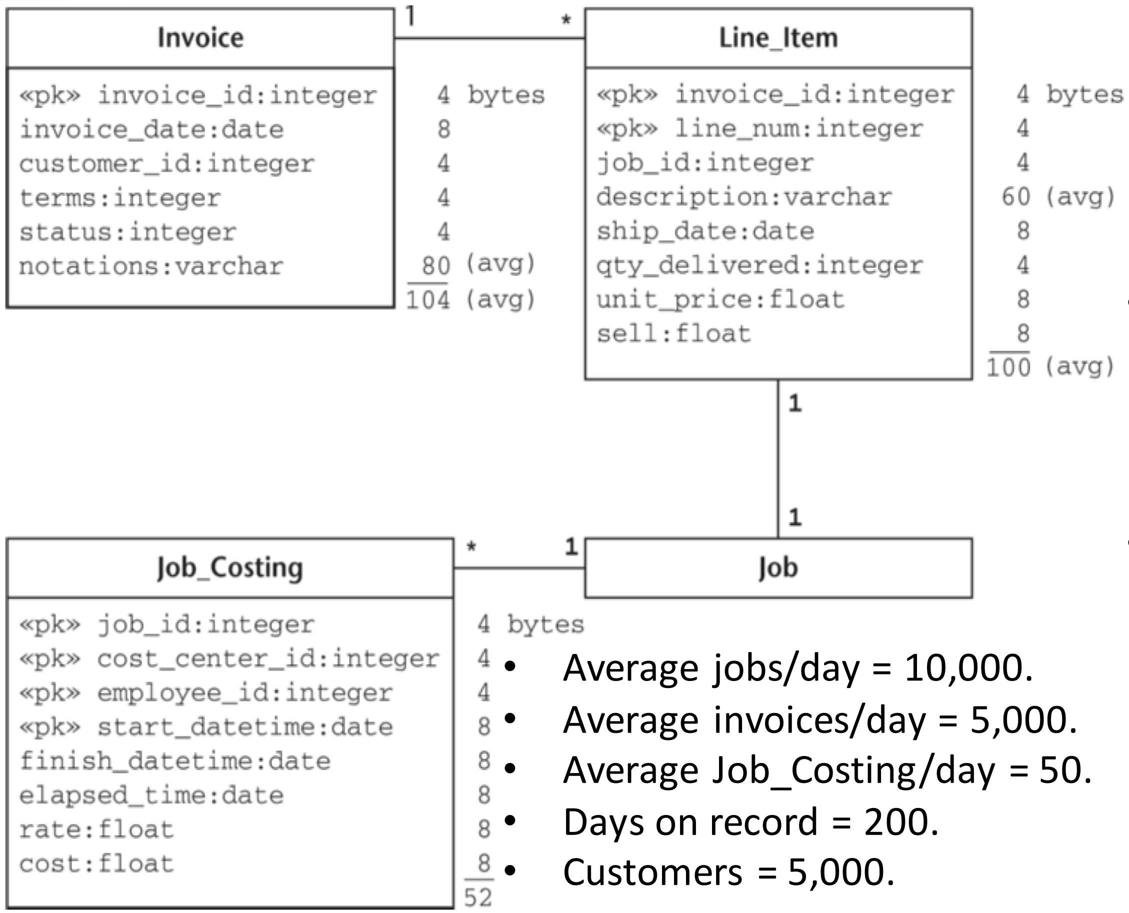


- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.

- Query I/O time:
  - Join cost (dedicated disk)
  - = 82,420 * 2.2 ms
  - = 181 seconds ≈ 3 minutes

- The Profit_by_Job calculations are:
  - Average rows per prefetch buffer
    - = floor((65,536 bytes/ buffer)/(28 bytes/row))
    - = 2,340.
  - Number of buffers
    - = ceiling((10,000*200) rows/(2,340 rows/buffer))
    - = 855.
  - Write cost (dedicated disk)
    - = 855*2.2 ms ≈ 2 seconds.
  - Profit_by_Job creation cost
    - = join cost + write cost
    - = 181 + 2 = 183 seconds.
  - Query I/O time (table scan, shared disk)
    - = 855*5.8 ms ≈ 5 seconds

Simple View Materialization

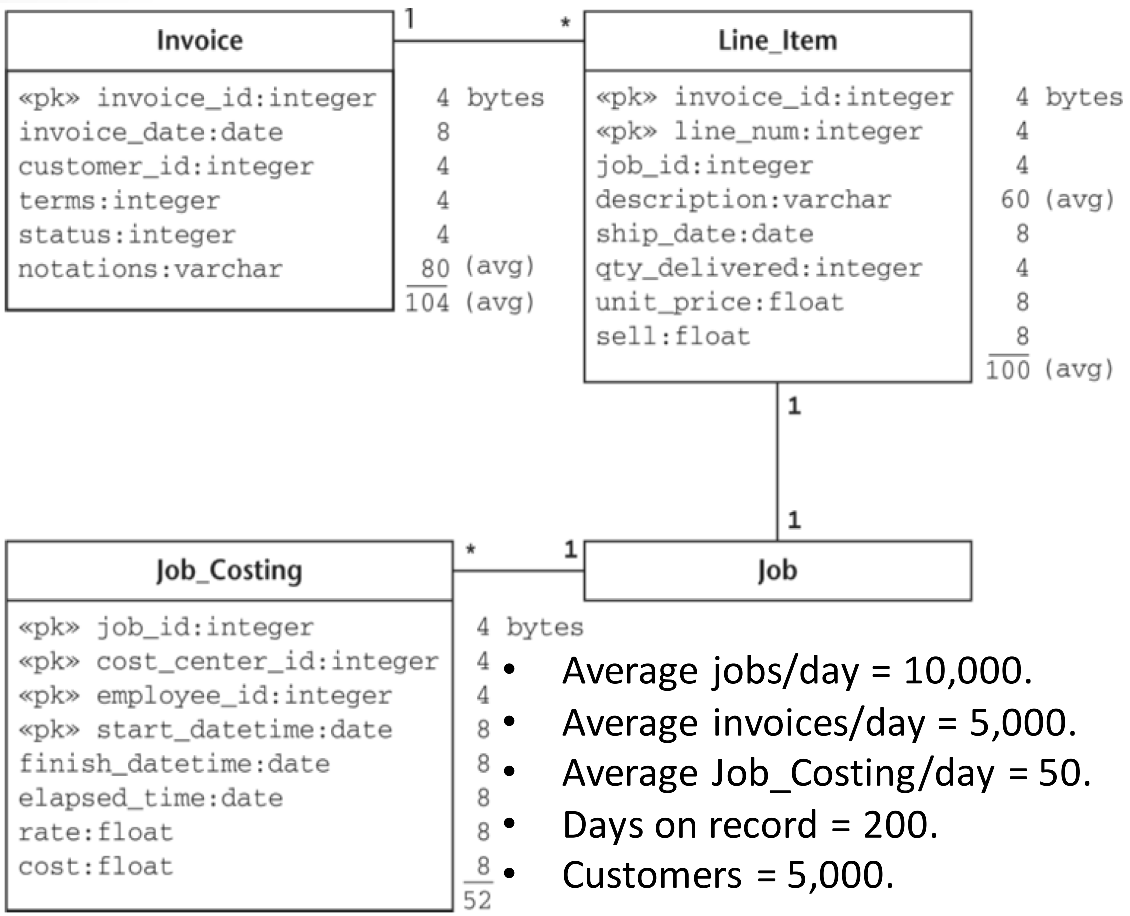


- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.

- Disk I/O time before Profit_by_Job:
  = query frequency * I/O time per query
  = (10 queries/day) * (478 sec/query)
  = 4,780 seconds (or about 1 hour, 20 min)

- Disk I/O time with Profit_by_Job:
  = creation cost + query frequency * I/O time per query
  = 183 sec createion cost + (10 queries/day) * (5 sec/query)
  = 233 seconds
  - The materialized view may be much smaller than the base tables, leading to large gains in disk I/O performance per query.
  - Frequent queries multiply the gain.

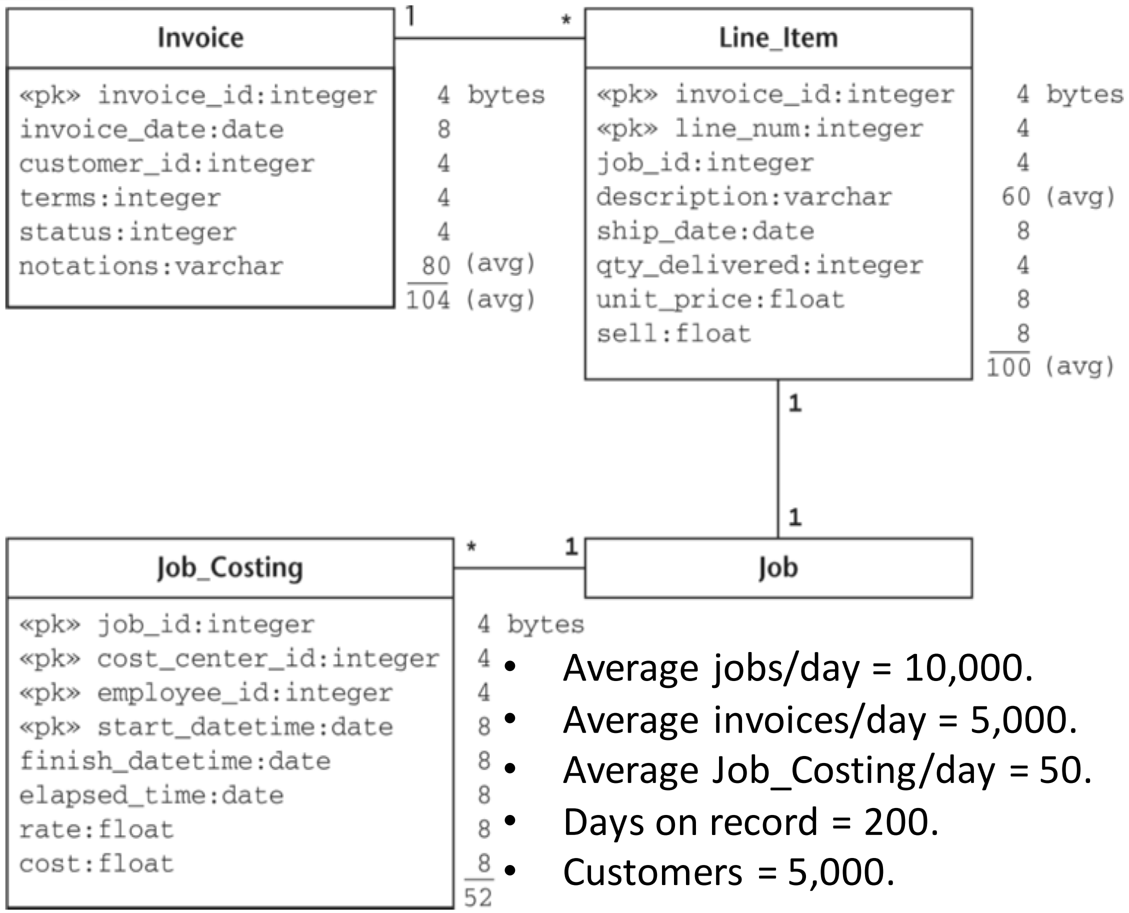
Exploiting Commonality



- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.

- Tracking profitability by invoice date (5 query/day).
- Tracking profitability by customer (3 query/day).

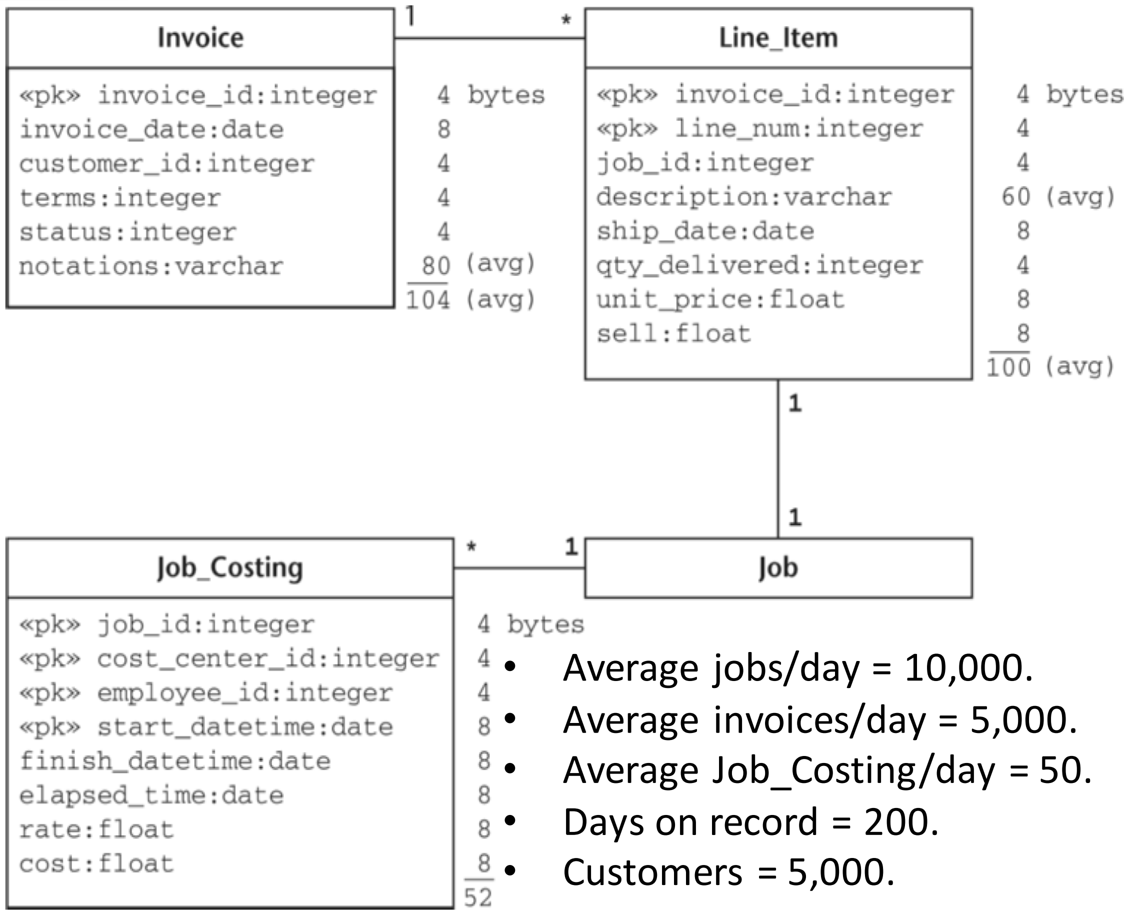
Tracking profitability by invoice date



- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.

```
SELECT i.invoice_date,
         sum(c.cost) AS cost,
         sum(li.sell)
         AS sell,
         sum(li.sell) - sum(c.cost)
AS                profit
FROM Invoice AS i, Job_Costing AS c,
         Line_Item AS li
WHERE i.invoice_id = li.invoice_id
AND c.job_id = li.job_id
GROUP BY i.invoice_date;
```

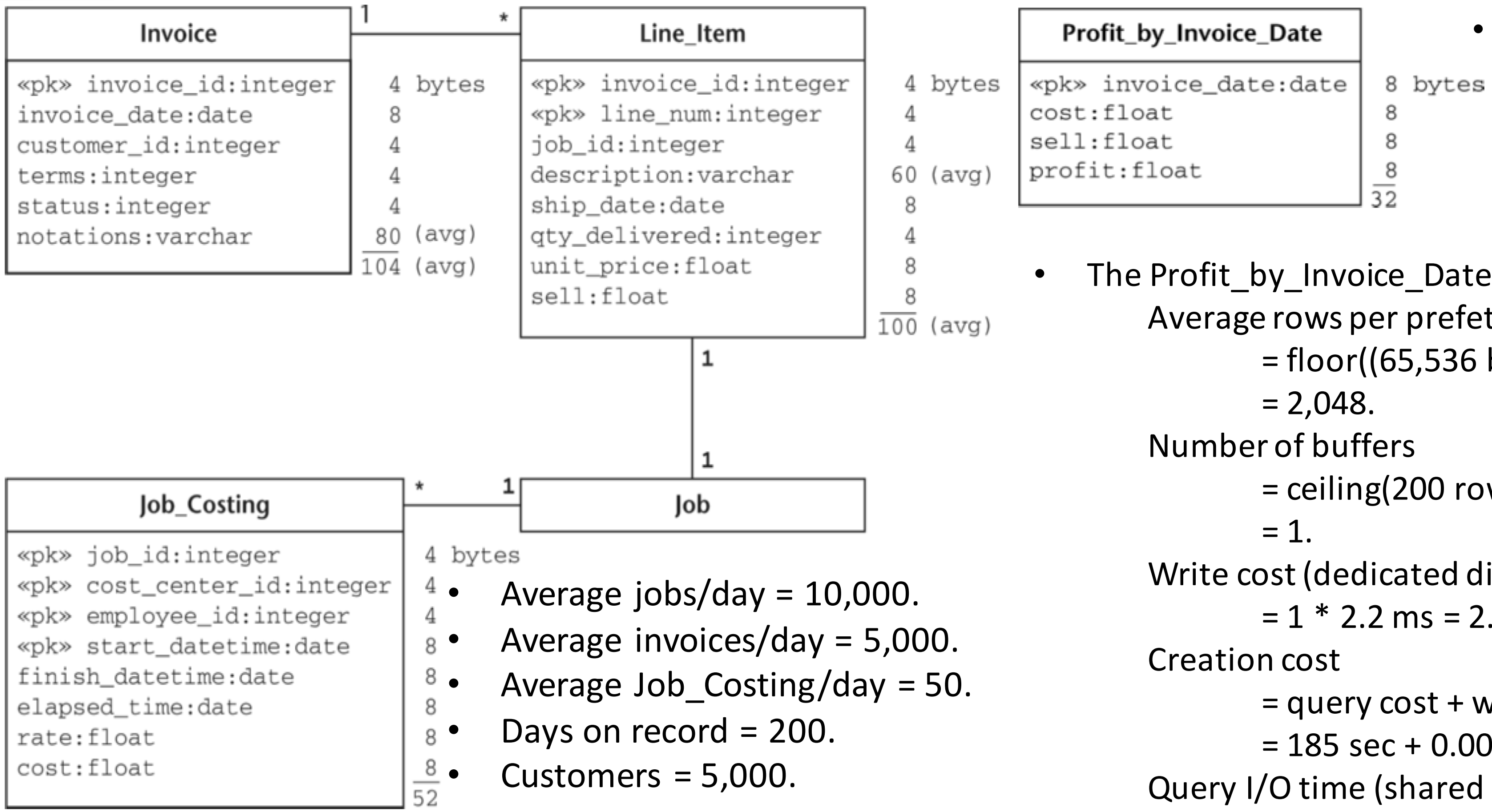
Tracking profitability by invoice date



- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.

- The Invoice calculations are:
  - Average rows per prefetch buffer
    - = floor((65,536 bytes/ buffer)/(104 bytes/row))
    - = 630.
  - Number of buffers
    - = ceiling((5000*200) rows/(630 rows/buffer))
    - = 1,588.
  - Query I/O time (shared disk)
    - = scan Line_Item + scan Invoice + scan Job_Costing
    - = (3,054 + 1,588 + 79,366 buffers) * 5.8 ms
    - = 84,008 * 5.8 ms
    - = 487 seconds.

Tracking profitability by invoice date

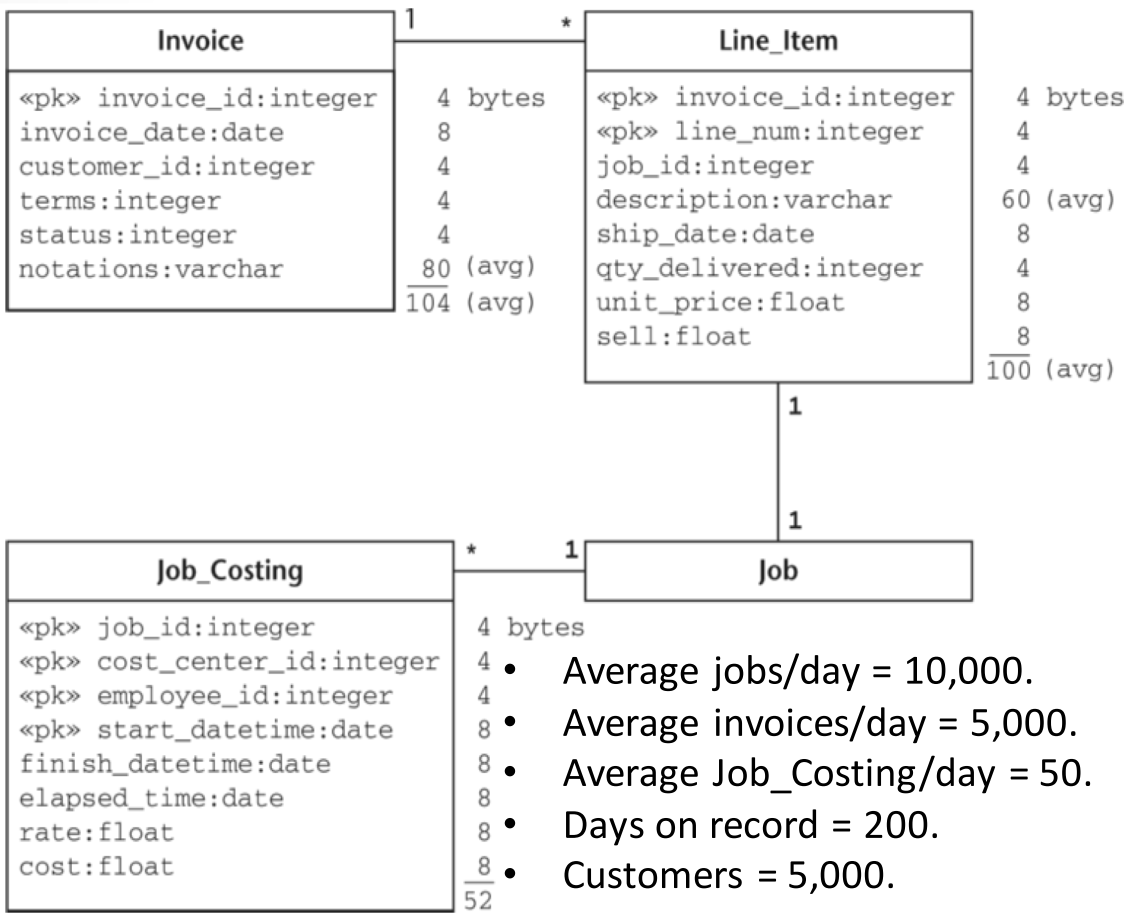


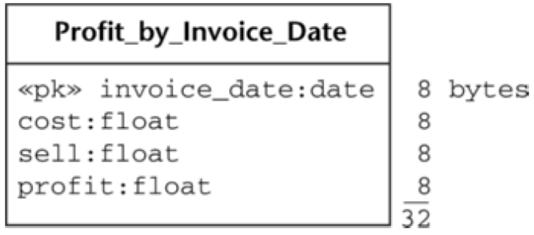
- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.

- Query I/O time:
  - = 84,008 * 2.2 ms
  - ≈ 185 seconds.

- The Profit_by_Invoice_Date calculations are:
  - Average rows per prefetch buffer
    - = floor((65,536 bytes/ buffer)/(32 bytes/row))
    - = 2,048.
  - Number of buffers
    - = ceiling(200 rows/(2,048 rows/buffer))
    - = 1.
  - Write cost (dedicated disk)
    - = 1 * 2.2 ms = 2.2 ms
  - Creation cost
    - = query cost + write cose
    - = 185 sec + 0.0022 sec ≈ 185 seconds.
  - Query I/O time (shared disk)
    - = 1 * 5.8 ms = 5.8 ms.

Tracking profitability by invoice date



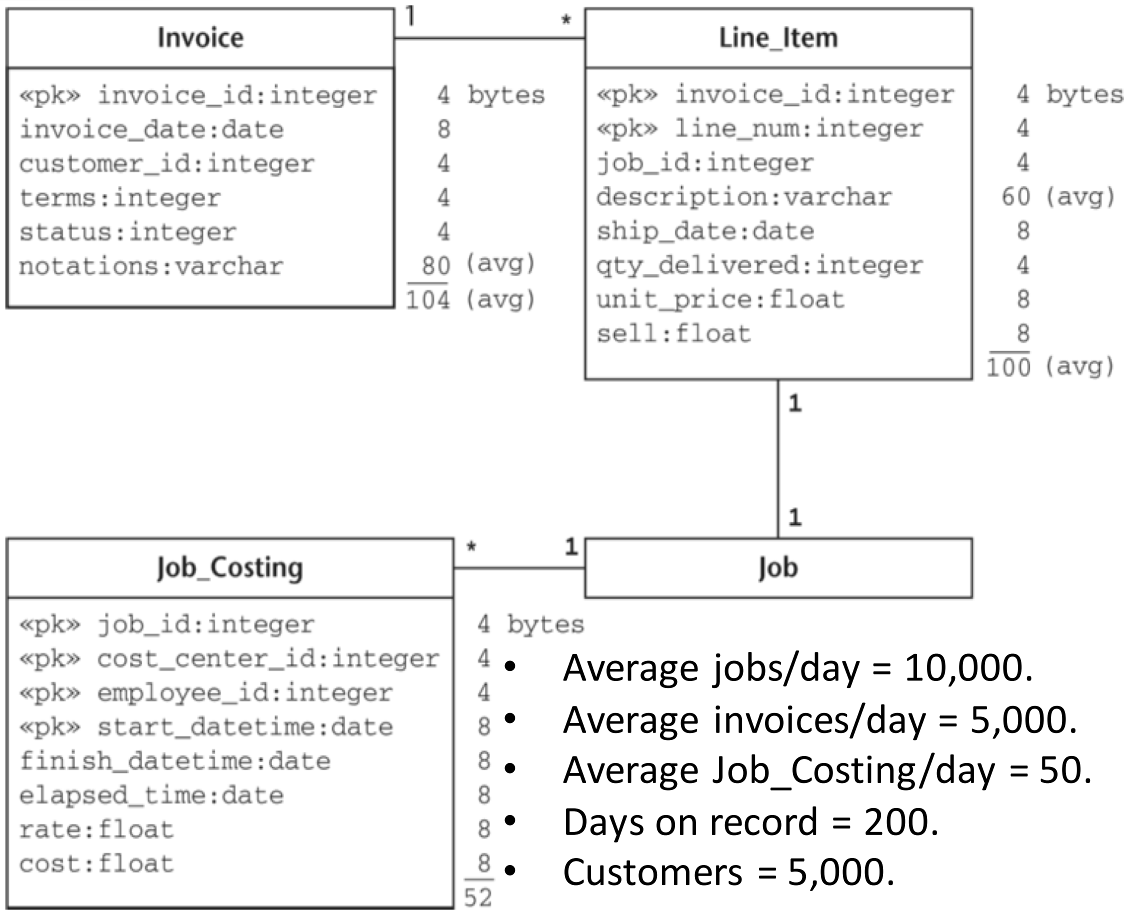


- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.

- Disk I/O time before Profit_by_Invoice_Date:
  - = query frequency * I/O time per query
  - = (5 queries/day) * (487 sec/query)
  - = 2,435 seconds (or about 40 min)

- Disk I/O time with Profit_by_Invoice_Date:
  - = creation cost + query frequency * I/O time per query
  - = 185 sec creation cost + (5 queries/day) * (5.8 ms/query)
  - ≈ 185 seconds

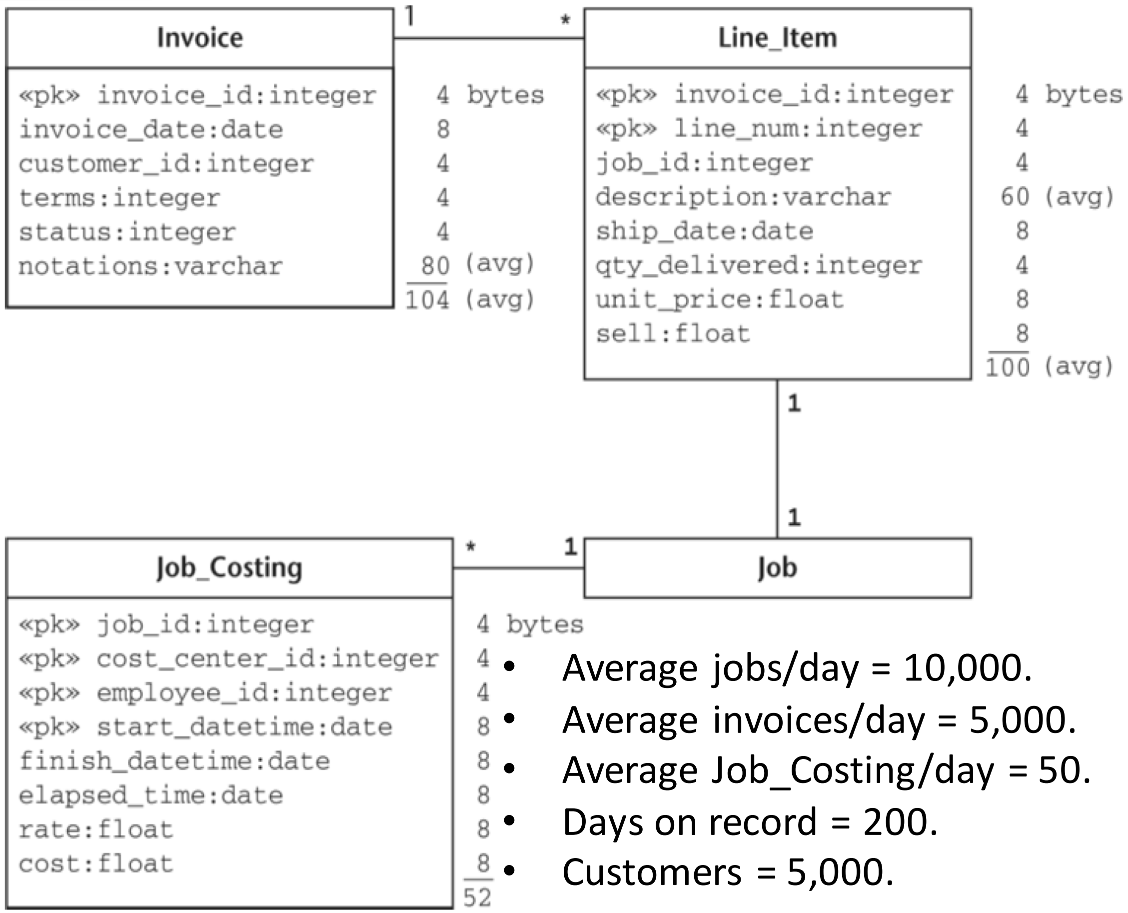
Tracking profitability by customer



- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.

```
SELECT i.customer_id,
          sum(c.cost)  AS cost,
          sum(li.sell)  AS sell,
          sum(li.sell)  - sum(c.cost)
AS                     profit
FROM Invoice AS i, Job_Costing AS c,
          Line_Item AS li
WHERE i.invoice_id = li.invoice_id
          AND c.job_id = li.job_id
GROUP BY i.customer_id;
```

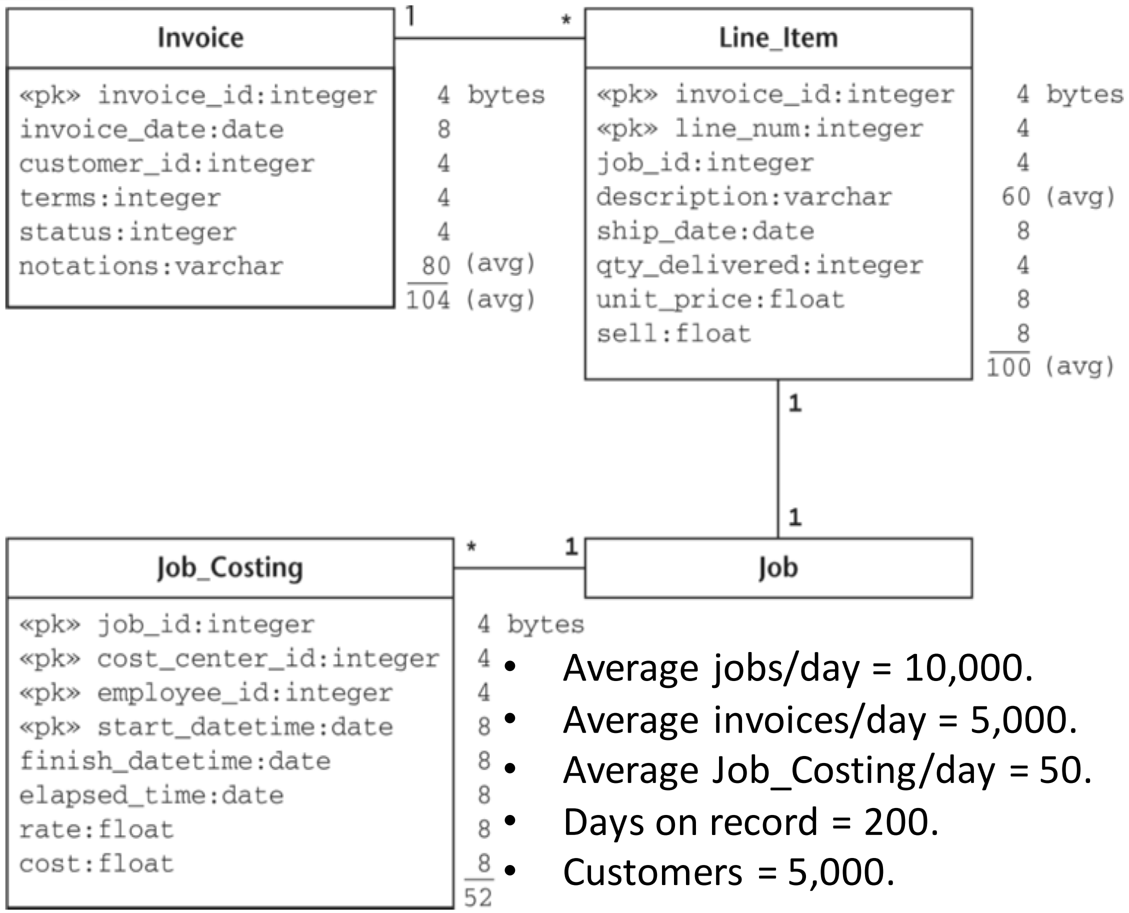
Tracking profitability by customer



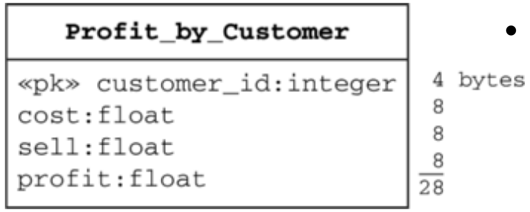
- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.

- The Invoice calculations are:
  - Average rows per prefetch buffer
    - = 630.
  - Number of buffers
    - = 1,588.
  - Query I/O time (shared disk)
    - = scan Line_Item + scan Invoice + scan Job_Costing
    - = 84,008 * 5.8 ms
    - = 487 seconds.

Tracking profitability by customer



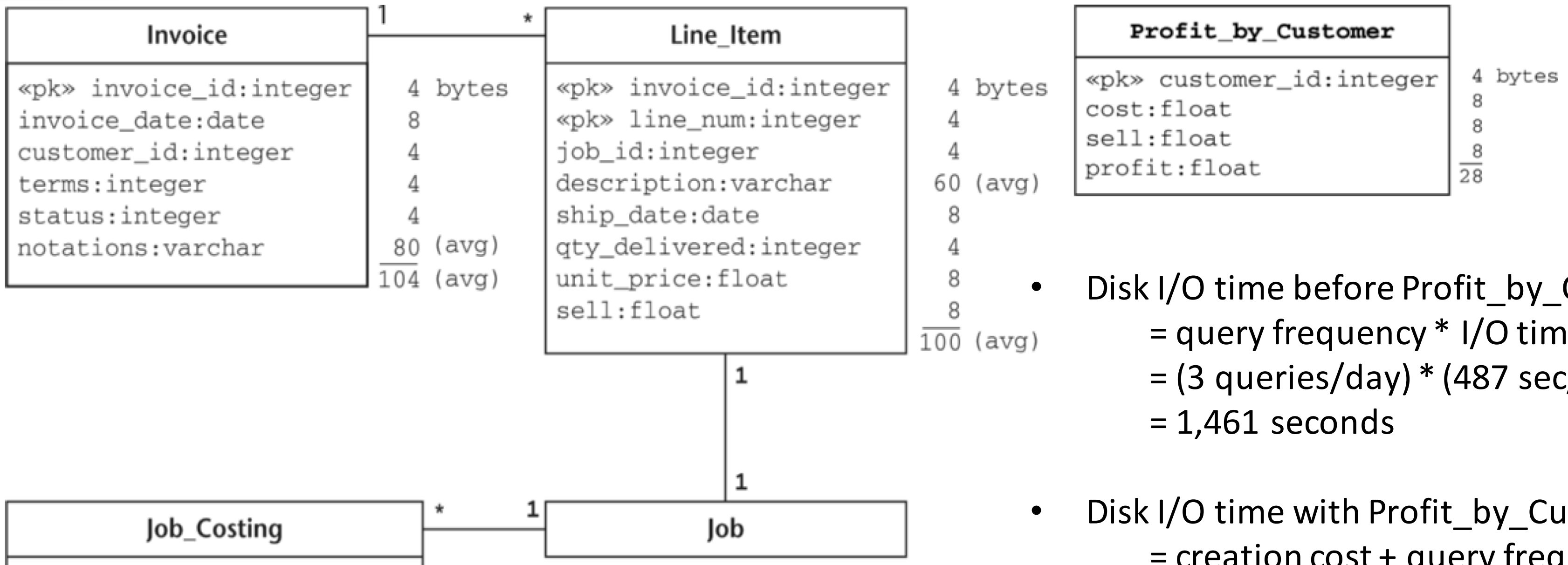
- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.

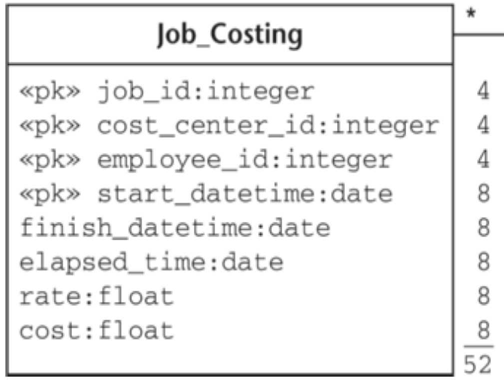


- Query I/O time:
  = 84,008 * 2.2 ms
  ≈ 185 seconds.

- The Profit_by_Customer calculations are:
  - Average rows per prefetch buffer
    - = floor((65,536 bytes/ buffer)/(28 bytes/row))
    - = 2,340.
  - Number of buffers
    - = ceiling(5000 rows/(2,340 rows/buffer))
    - = 3.
  - Write cost (dedicated disk)
    - = 3 * 2.2 ms = 6.6 ms
  - Creation cost
    - = query cost + write cose
    - = 185 sec + 0.0066 sec ≈ 185 seconds.
  - Query I/O time (shared disk)
    - = 3 * 5.8 ms = 17.4 ms.

Tracking profitability by customer





- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.

- Disk I/O time before Profit_by_Customer:
  = query frequency * I/O time per query
  = (3 queries/day) * (487 sec/query)
  = 1,461 seconds

- Disk I/O time with Profit_by_Customer :
  = creation cost + query frequency * I/O time per query
  = 185 sec creation cost + (3 queries/day) * (0.017 sec/query)
  ≈ 185 seconds

Exploiting Commonality

Profit_by_Job	
«pk» job_id:integer	4 bytes
cost:float	8
sell:float	8
profit:float	8
	28

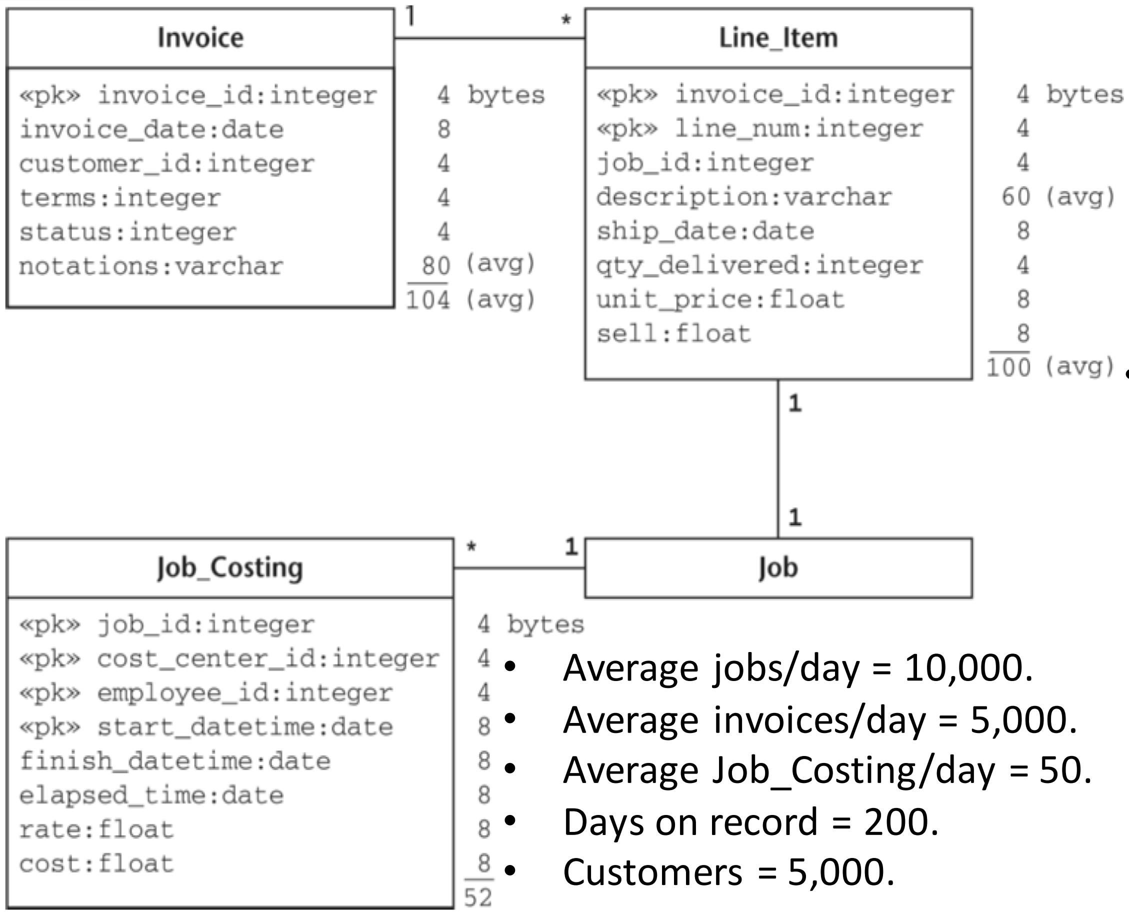
Profit_by_Customer	
«pk» customer_id:integer	4 bytes
cost:float	8
sell:float	8
profit:float	8
	28

Profit_by_Invoice_Date	
«pk» invoice_date:date	8 bytes
cost:float	8
sell:float	8
profit:float	8
	32

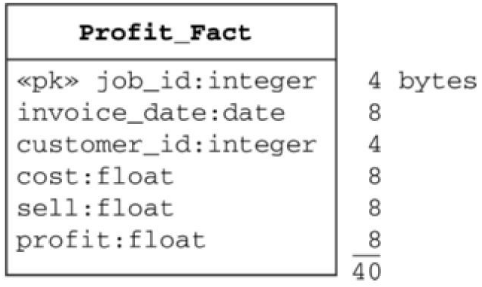
Profit_Fact	
«pk» job_id:integer	4 bytes
invoice_date:date	8
customer_id:integer	4
cost:float	8
sell:float	8
profit:float	8
	40

```
SELECT c.job_id, i.invoice_date,
          i.customer_id,
          sum(c.cost) AS cost,
          sum(li.sell) AS sell,
          sum(li.sell) - sum(c.cost) AS profit
FROM Invoice AS i, Job_Costing AS c,
          Line_Item AS li
WHERE i.invoice_id = li.invoice_id
          AND c.job_id = li.job_id
GROUP BY c.job_id, i.invoice_date,
          i.customer_id;
```

Exploiting Commonality



- Average jobs/day = 10,000.
- Average invoices/day = 5,000.
- Average Job_Costing/day = 50.
- Days on record = 200.
- Customers = 5,000.



- Query I/O time:
  = 84,008 * 2.2 ms
  ≈ 185 seconds.

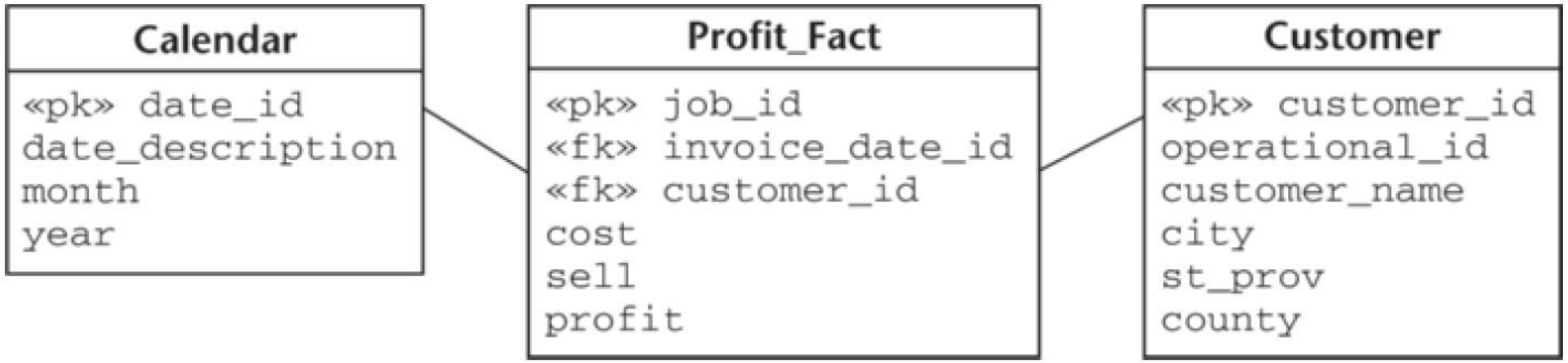
- The Profit_Fact calculations are:
  - Average rows per prefetch buffer
    = floor((65,536 bytes/buffer)/(40 bytes/row))
    = 1,638.
  - Number of buffers
    = ceiling((10,000*200)rows/(1,638 rows/buffer))
    = 1,222.
  - Write cost (dedicated disk)
    = 1,222 * 2.2 ms ≈ 3 seconds.
  - Creation cost
    = query cost + write cose
    ≈ 185 sec + 3 sec ≈ 188 seconds.
  - Query I/O time (shared disk)
    = 1,222 * 5.8 ms = 7 seconds.

Exploiting Commonality

Profit_Fact	
«pk» job_id:integer	4 bytes
invoice_date:date	8
customer_id:integer	4
cost:float	8
sell:float	8
profit:float	8
	40

- Disk I/O time using base tables:
  = (4,780 + 2,435 + 1,461 sec)/day
  = 8,676 sec/days.
- Disk I/O time using Profit_by_Job, Profit_by_Invoice_Date, Profit_by_Customer:
  = (233 + 185 + 185 sec)/day
  = 603 sec/day
- Disk I/O time using Profit_Fact:
  = creation cost + (query frequency* I/O time per query)
  = 188 sec creation cost
  + (10 profit by job queries/day)*(7 sec/query)
  + (5 profit by date queries/day)*(7 sec/query)
  + (3 profit by customer queries/day)*(7 sec/query)
  ≈ 314 sec/day.

Exploiting Group and Generalization



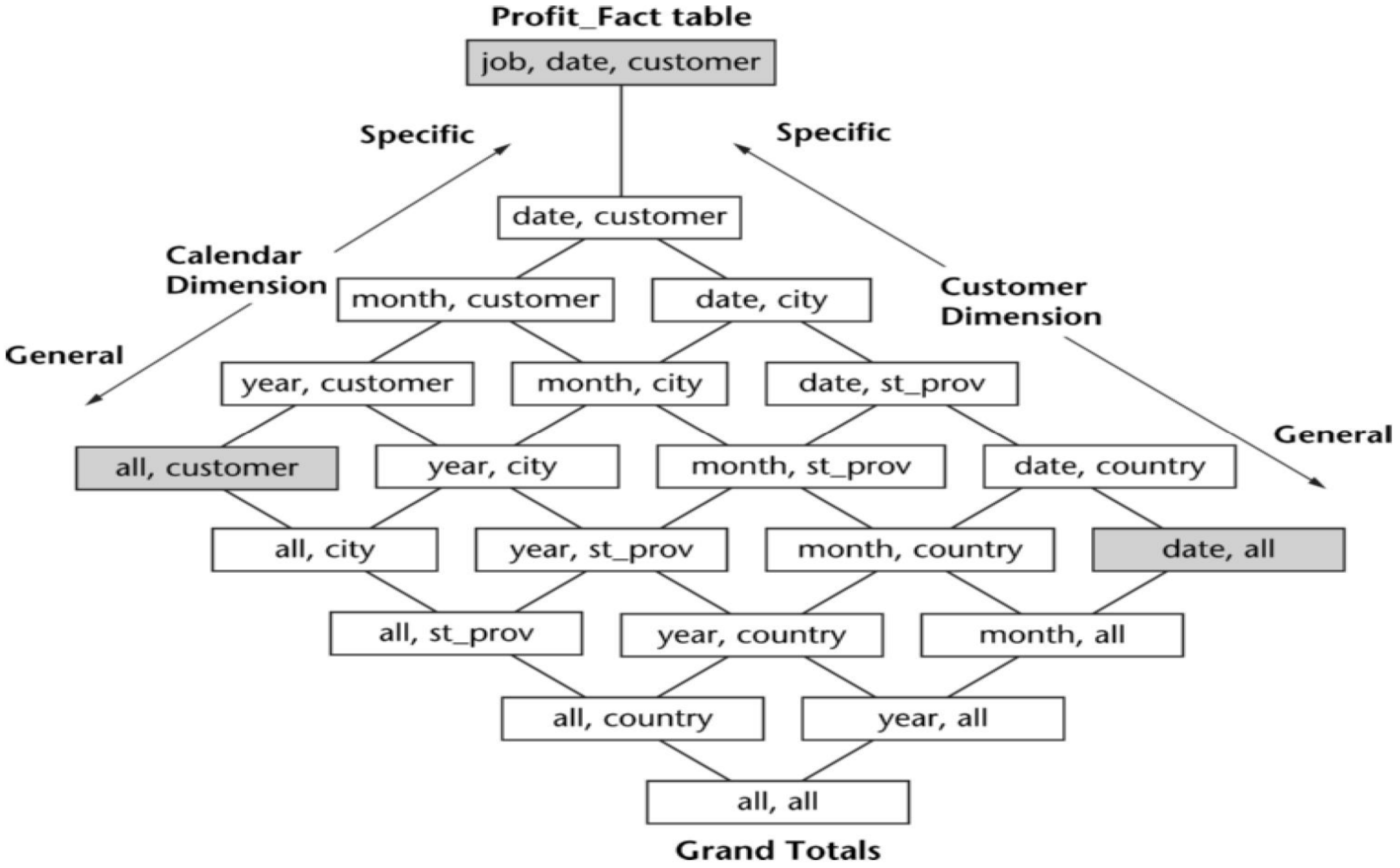
This is the query to obtain profitability information at the monthly level for 2006:

```
SELECT month, sum(cost), sum(sell),
          sum(profit)
FROM Calendar AS c, Profit_Fact AS p
WHERE year = 2006
          AND c.date_id = p.date_id
GROUP BY month
ORDER BY month;
```

This is the query to obtain data at the state level by year:

```
SELECT country, st_prov, year,
          sum(cost), sum(sell),
          sum(profit)
FROM Calendar AS cal, Profit_Fact AS    p,
Customer AS cust
WHERE cal.date_id = p.date_id
          AND cust.customer_id =
p.customer_id
GROUP BY country, st_prov, year
ORDER BY country, st_prov, year;
```

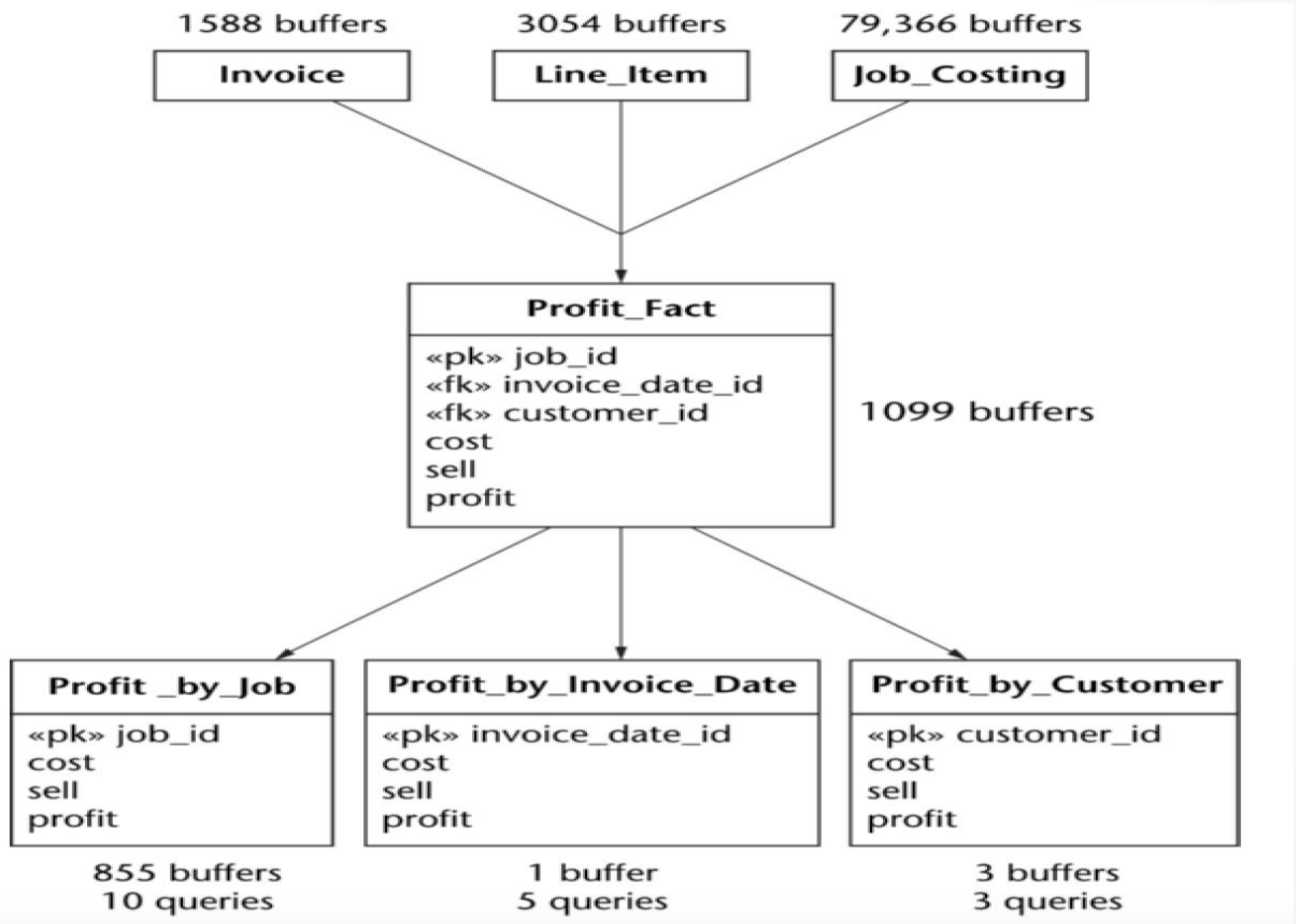
Exploiting Group and Generalization



Product graph in Calendar and Customer dimensions.

Resource Considerations

- The number of materialized views.
- The extra disk space required.
- The length of the available update windows.



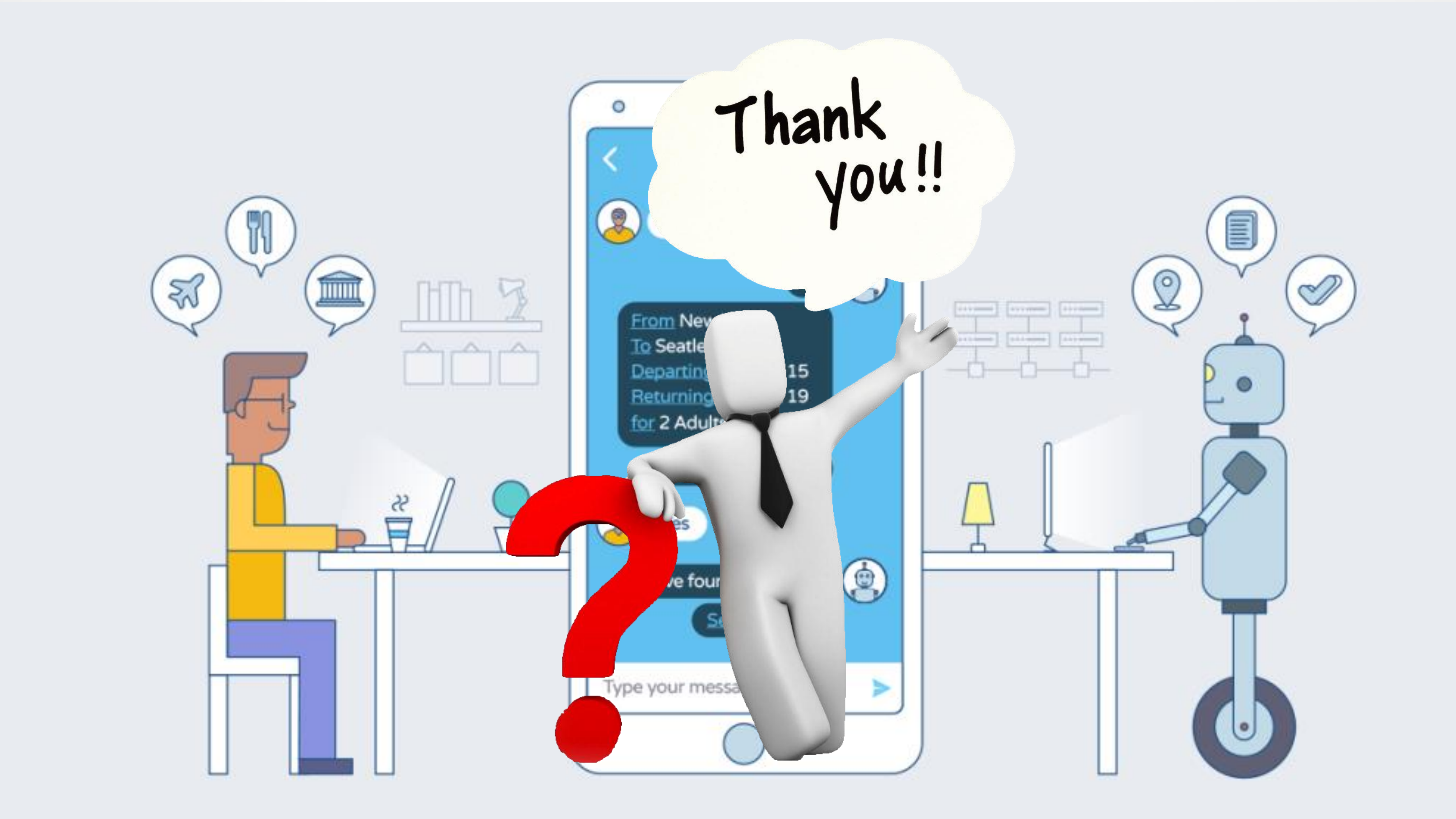
A simplified lattice

Tips and insights for database professionals

- Utilizing materialized view can bring marked improvement in both total disk I/O and query response.
- Good candidates to consider for materialization are the natural view of frequent queries and also common ancestors of those views.
- Star schemas can make a materialized view applicable to a large family of queries, thereby multiplying the gain for the given resources.
- Lattice structure diagrams can facilitate the selection of materialized views and also the planning of data update paths through the lattice.

Tips and insights for database professionals

- Decide on the update strategy for each materialized view.
- Set a limit on the number of views you are willing to design and maintain.
- Decide on a limit for the amount of disk space available for materialized views.
- Materialized views need indexing too!
- Help the query compiler find matching materialized views.
- Avoid problematic materialized view designs that make routing hard.


Thank you!!
Type your messa