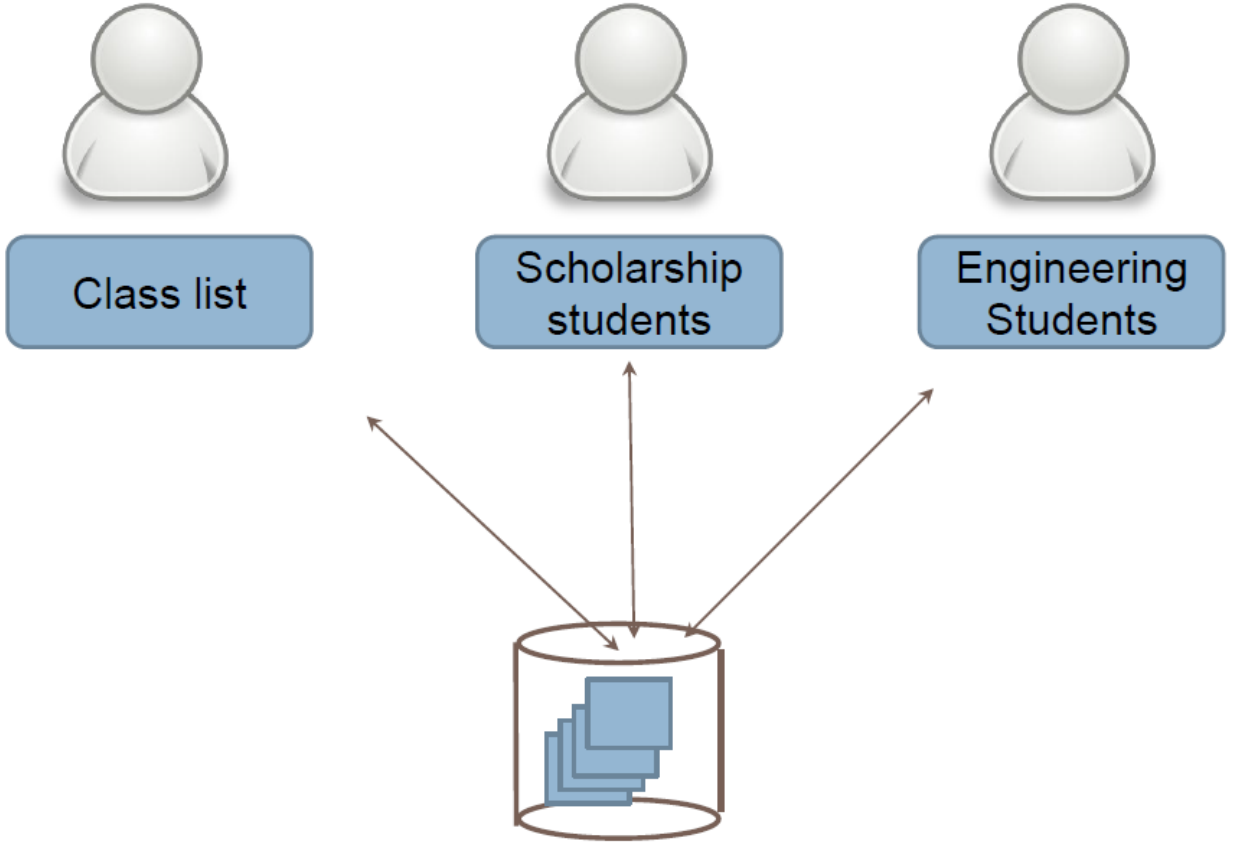
Views and Indexes
CSC 343
Winter 2018

MICHAEL LIUT (MICHAEL.LIUT@UTORONTO.CA)
DEPARTMENT OF MATHEMATICAL AND COMPUTATIONAL SCIENCES
UNIVERSITY OF TORONTO MISSISSAUGA

UNIVERSITY OF TORONTO MISSISSAUGA

Scenario



Views

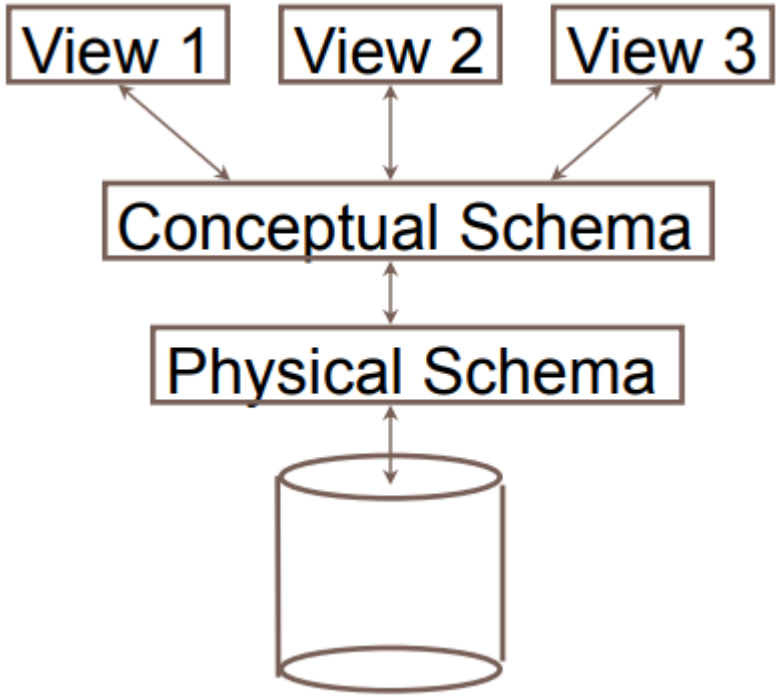
More often than not, it is not desirable for all users to see the entire data instance. This could be for many reasons, some of which are:

- Security
- Privacy
- National Governance (i.e. Laws)

A **view** provides a mechanism to hide certain data from the view of certain users.

Levels of Abstraction

1. Database
2. Physical Schema
   - Describing the files and index(es) used.
3. Conceptual (Logical) Schema
   - Defining the logical structure.
4. Views
   - Describe how users can see the data.



Views

A **view** is a relation defined in terms of stored tables (i.e**. base tables**) and other views

Two types:

1. **Virtual**: not stored in the database; it is a query for constructing the relation each time they are accessed.
2. **Materialized**: actually constructed and stored; updated periodically depending on the query definition.

Declaring Views

Declare by:

CREATE [MATERIALIZED] VIEW <name> AS <query>;

the DEFAULT view is VIRTUAL

- A view name
- A possible list of attribute names
  - e.g., when arithmetic operations are specified or when we want the names to be different from the attributes in the base relations.
- A query to specify the view contents

Example: View Definition

CanDrink(drinker,beer) is a view "containing" the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS
        SELECT  drinker, beer
        FROM    Frequents, Sells
        WHERE   Frequents.bar = Sells.bar;
```

Example: Accessing a View

Query a view as if it were a *base table*.

- Note that there is a limited ability to modify views; just as if it were an underlying base table.

Example Query:

```
SELECT beer FROM CanDrink
WHERE drinker = 'Sally';
```

Another Example

View Synergy(drinker, beer, bar) such that the bar serves the beer, the drinker frequents the bar, and the drinker likes the beer.

```
CREATE VIEW Synergy AS
SELECT Likes.drinker, Likes.beer, Sells.bar
FROM Likes, Sells, Frequents
WHERE Likes.drinker = Frequents.drinker
      AND Likes.beer = Sells.beer
      AND Sells.bar = Frequents.bar;
```

Pick one copy of each attribute

Natural join of Likes, Sells, and Frequents

Updates on Views

Generally, it is impossible to modify a virtual view; because it does not exist.

Q: Can't we 'translate' updates on views into 'equivalent' updates on base tables?

A: Not always! In fact, not often, as most systems prohibit the majority of view updated. For example, we cannot insert into Synergy as it is a virtual view that does not meet the criteria of a single relation in the FROM clause.

Interpreting a View Insertion

We could try to translate the (drinker, beer, bar) triple, using a trigger (e.g. INSTEAD OF), into three insertions of projected pairs. One for each of Likes, Sells, and Frequents.

Sells.price will have to be NULL.
There is not always a unique translation.

```
CREATE TRIGGER ViewTrig
    INSTEAD OF INSERT ON Synergy
    REFERENCING NEW ROW AS n
    FOR EACH ROW
    BEGIN
        INSERT INTO LIKES VALUES(n.drinker, n.beer);
        INSERT INTO SELLS(bar, beer) VALUES(n.bar, n.beer);
        INSERT INTO FREQUENTS VALUES(n.drinker, n.bar);
    END;
```

Materialized Views

Materialized: actually constructed and stored (keeping a temporary table).

- If a view is used frequently enough, it may be more efficient to materialize it!

- Concerns: maintaining correspondence between the base table and the view when the base table is modified in any manner. This is computationally costly.

- Strategy: incremental update or periodic reconstruction.

Materialized Views

Strategy: incremental update or periodic reconstruction.

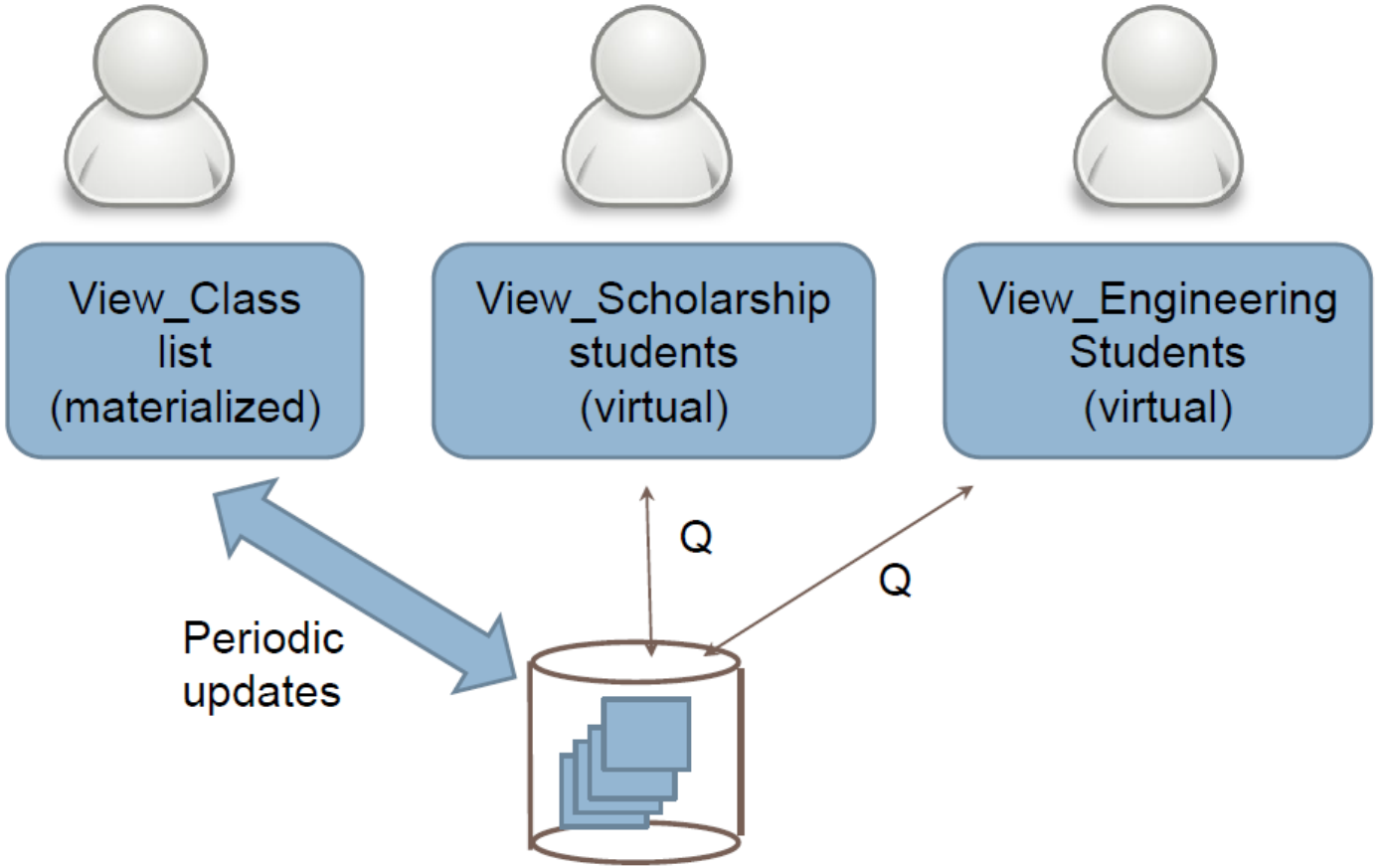
1. Incremental Update

   Insertions, deletions, and updates to a base table can be implemented in a join view by a small number of queries to the base tables followed by modification statements on the materialized view.

2. Periodic Reconstruction

   Reconstructing the materialized view which is “out-of-date”.

Example


View_Class list (materialized)
View_Scholarship students (virtual)
View_Engineering Students (virtual)
Q
Q
Periodic updates

Example: Class Mailing List

The class mailing list for **csc343student** is in effect a materialized view of class enrollment.

- This is updated periodically as you can enroll and drop the course.
- An email can be sent after you've enrolled, however, you may not receive it as the email list may not be updated.

Insertion into a materialized view is normally followed by insertion into the base table.

Materialized View Updates

Update on a single view without aggregate operations.

- UPDATE may map to an update on the underlying base table; as seen in the majority of SQL implementations.

Views involving JOINs.

- UPDATE may map to an update on the underlying base relations; not always possible.

How about a Data Warehouse?

Recall: Walmart from the Pop-Tarts example, and how Walmart stores every transaction they process.

Overnight, the sales for the day are used to update a ***materialized view*** representing the sales for that day, maybe even updating a yearly total.

- This view is used by analysts to perform *Data Analytics* on trends/patterns in the data to improve sales and ensure proper inventory stock.

The activity of examining data for patterns or trends is called **OLAP** (**O**n-**L**ine **A**nalytic **P**rocessing). Generally involving highly complex queries that use one or more aggregations.

Indexes in General

Index: data structure used to speed access to tuples of a relation, given values of one or more attributes.

- It can be expensive to scan all tuples to find only those that match a given condition.

These can be ***Hash Tables***, but in a DBMS it is always a balanced search tree with giant nodes (a full disk page) called a ***B-tree***.

Declaring Indexes

There is no SQL standard!

Typical Syntax:

```
CREATE INDEX BeerIndex ON Beers(manf);
CREATE INDEX SellIndex ON Sells(bar, beer);
```

Using Indexes

Given a value *v*, the index takes us to only those tuples that have *v* in the attribute(s) of the index.

Example: use BeerIndex and SellIndex to find the prices of beers manufactures by Pete's and sold by Joe. ***[Next Slide]***

Using Indexes

Example: use BeerIndex and SellIndex to find the prices of beers manufactures by Pete's and sold by Joe.

```
SELECT price FROM Beers, Sells
WHERE manf = 'Pete"s' AND
    Beers.name = Sells.beer AND
    bar = 'Joe"s Bar';
```

1. Use BeerIndex to get all the beers made by Pete's.
2. Then use SellIndex to get prices of those beers, with bar = 'Joe"s Bar'

Database Tuning

A major problem in making a database ‘run fast’ is deciding which indexes to create.

- **Pro**: an index speeds up queries that can use it.
- **Con**: an index slows down all modifications on its relation because the index must be modified too.

Often, the most useful index we can put on a relation is an index on its key (as it will be utilized frequently).

Example: Database Tuning

Suppose the only things done with the beers database are as follows:

1. Insert new facts into a relation (10%).
2. Find the price of a given beer at a given bar (90%).

Then **SellIndex** on **Sells(bar, beer)** would be wonderful, but **BeerIndex** on **Beers(manf)** would be harmful.

Database Tuning Advisors

Hand tuning is too hard, so often DBAs use an advisor which gets a ***query load***:

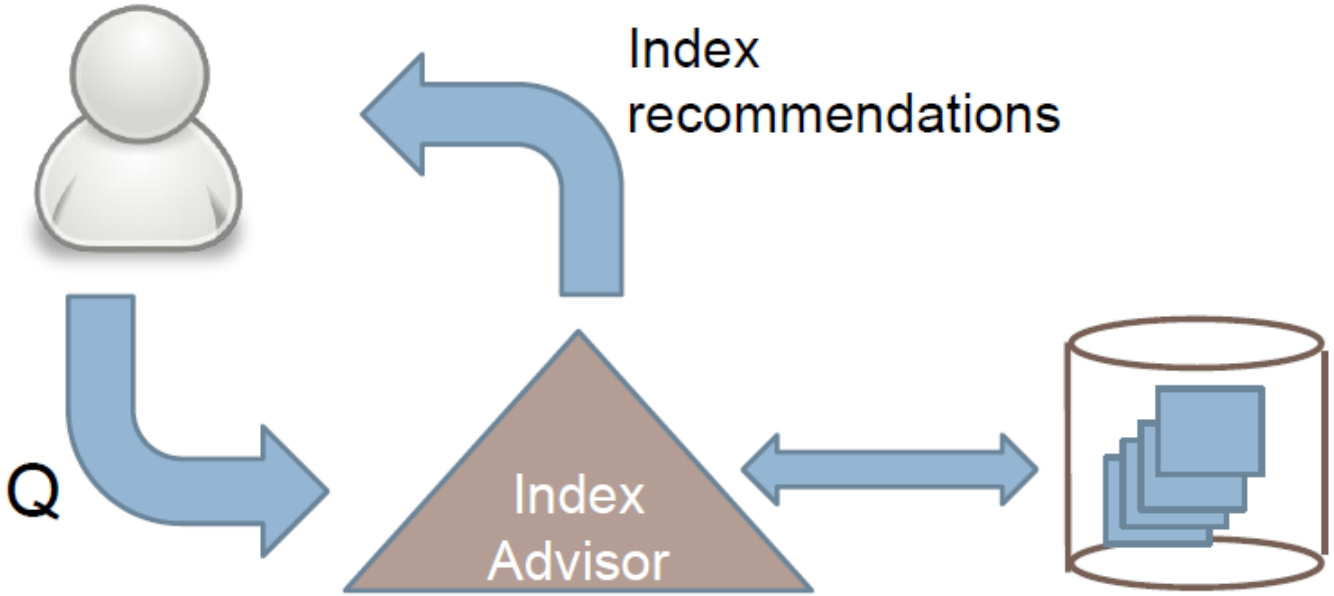
1. select random queries from past executed queries on the database; or
2. the designer provides a sample work.

The advisor generates candidate indexes and evaluates them.

- Feed the sample query to the optimizer; assuming that only this index is available.
- Measure the improvement/degradation in the average running time of the queries.

Database Tuning Advisors

The advisor generates candidate indexes and evaluates them on the workload! Then measures the improvement/degradation in the average running time of the queries.



An Index

A data structure that organizes the records via trees or hashing

- They speed up searches for a subset of records based on values in certain ("search key") fields.

Given a value *v*, the index takes us to only those tuples that have *v* in the attribute(s) of the index.

- e.g., use BeerIndex (on manf) and SellIndex (on bar, beer) to find the prices of beers manufactures by Pete's and sold by Joe.

Tree-based Indexes

One of the most basic ideas is to create a file with one record per page, of the form *<key on page, pointer to page>* as index entries.

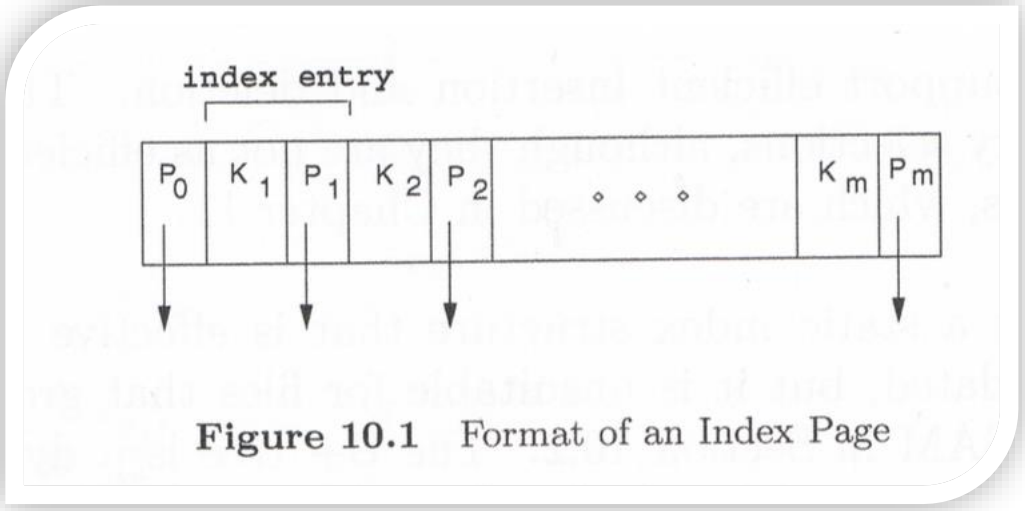


Figure 10.1 Format of an Index Page

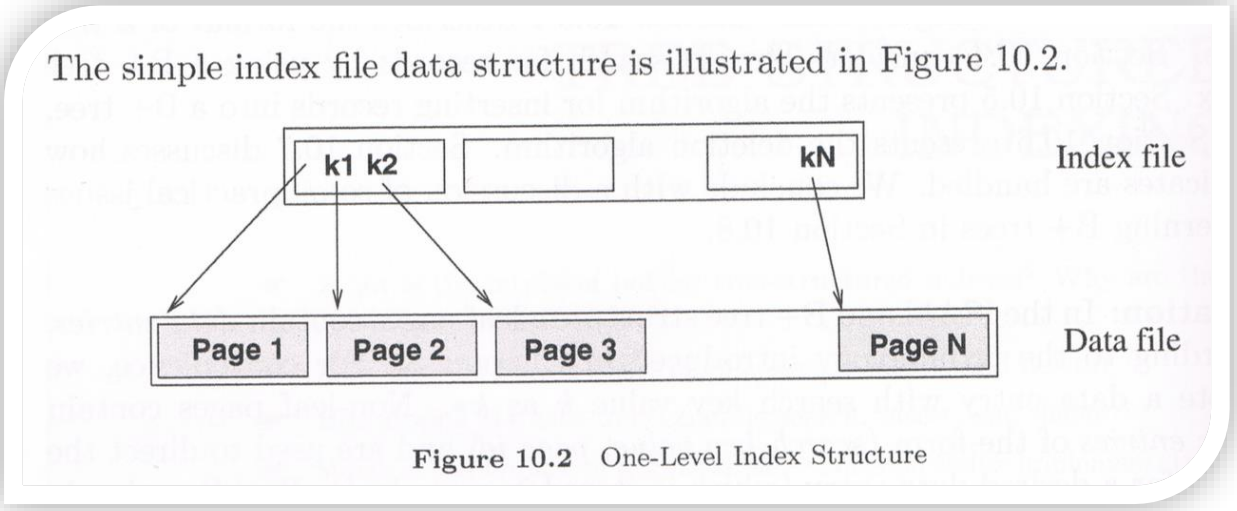


Figure 10.2 One-Level Index Structure

Indexed Sequential Access Method

ISAM is a data structure which retains its data entries in the leaf pages of the tree and additional *overflow* pages chained to some leaf page.

This structure is static in nature; except in the *overflow* pages.

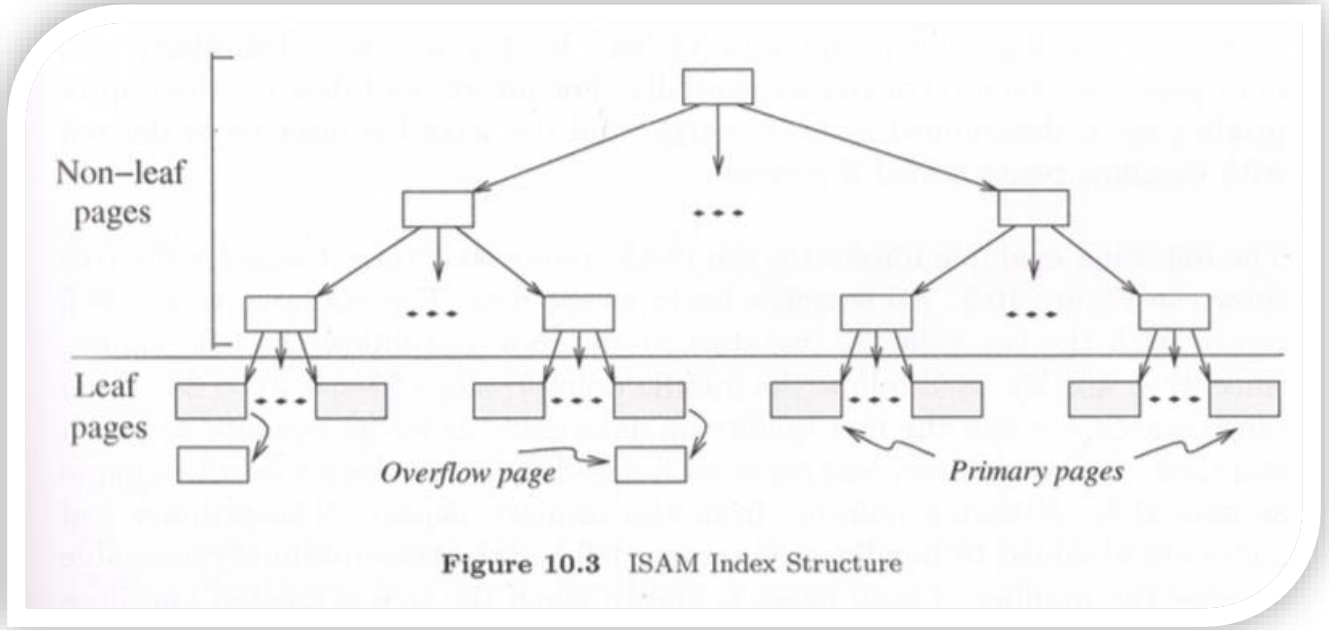
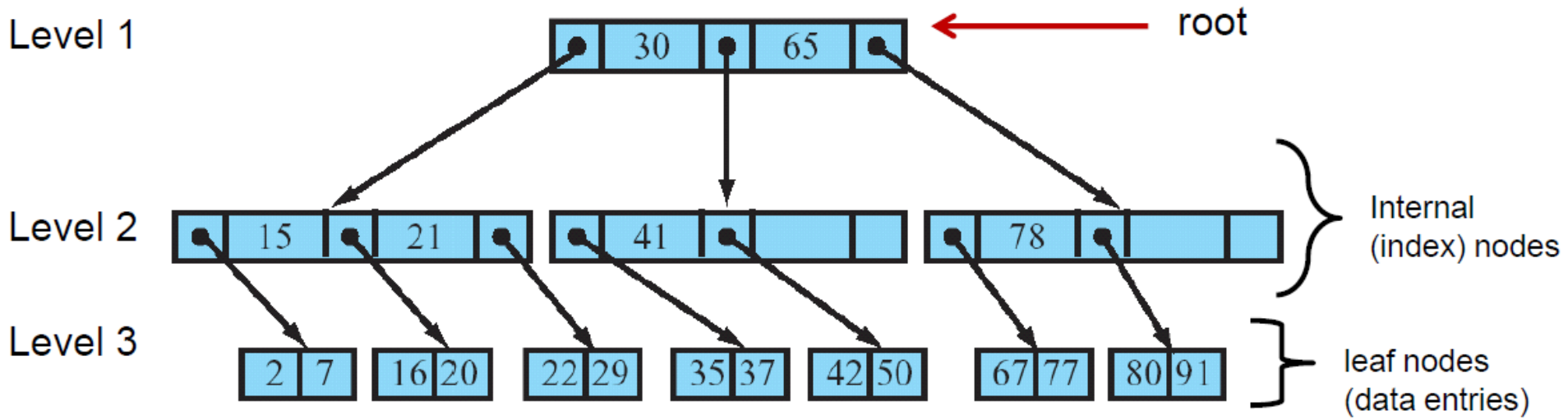


Figure 10.3 ISAM Index Structure

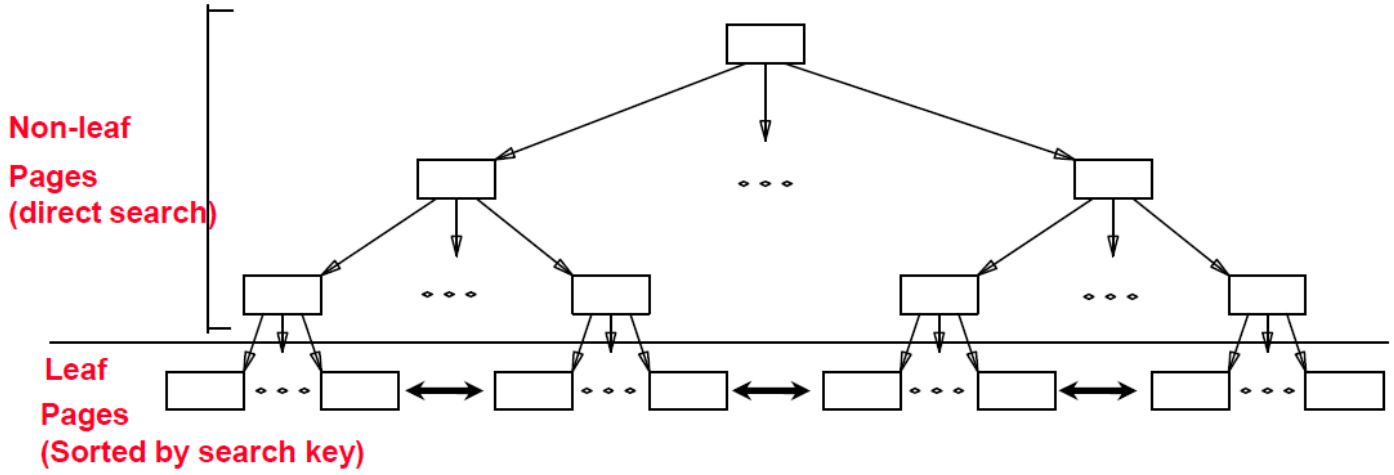
B+ Tree

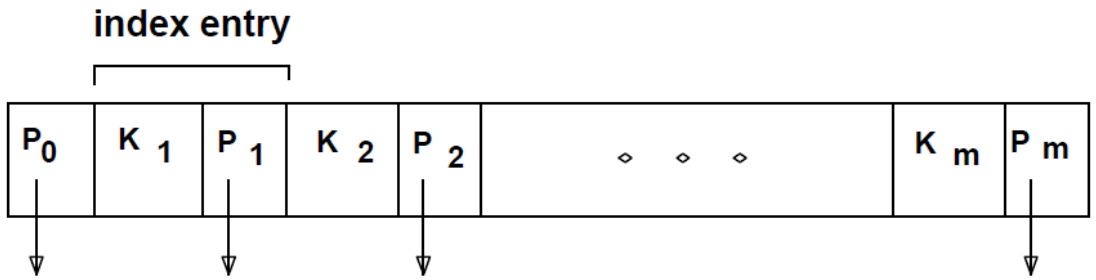
- The B+ tree structure is the most common index type in databases today.
- As index files can be quite large, often stored on disks, partially loaded into memory as needed.
- Each node is at least 50% full.



B+ Tree Index

Supports equality and range-searches efficiently!



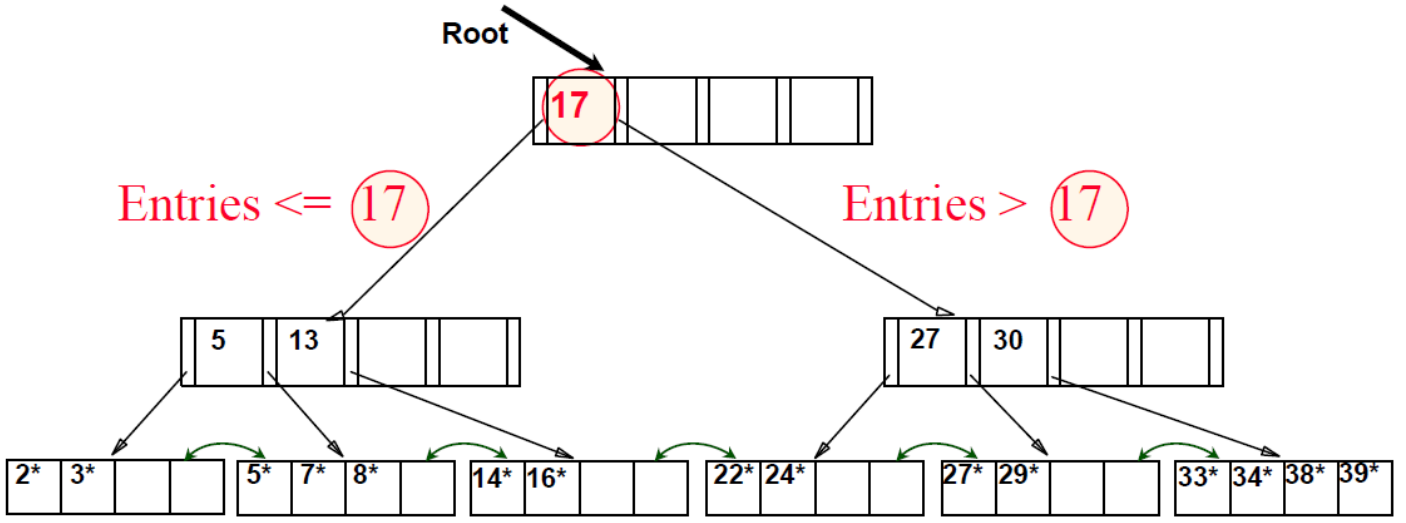


Example: B+ Tree Search/Modify

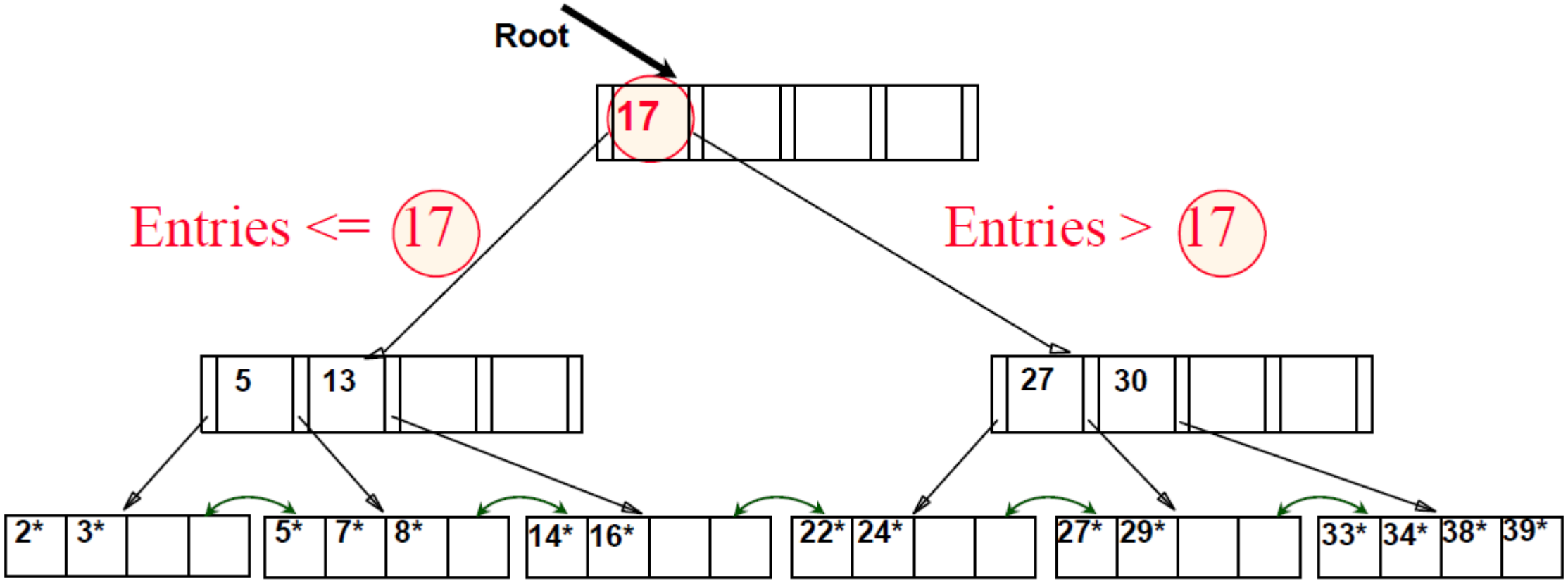
1. Let's say that we are looking for 28*? 29*? All > 15* AND < 30*

2. Insert/Delete:

   Find a data entry in a leaf node and modify it.



Example: B+ Tree Search



B+ Tree: Inserting a Data Entry

1. Find the correct leaf *L*.
2. Put data entry onto *L*.
   i. If *L* has enough space, done!
   ii. Else, must split *L* (into *L* and a new node L_2).
      i. Redistribute entries evenly, copy up middle key.
      ii. Insert index entry pointing to L_2 into parent of *L*.

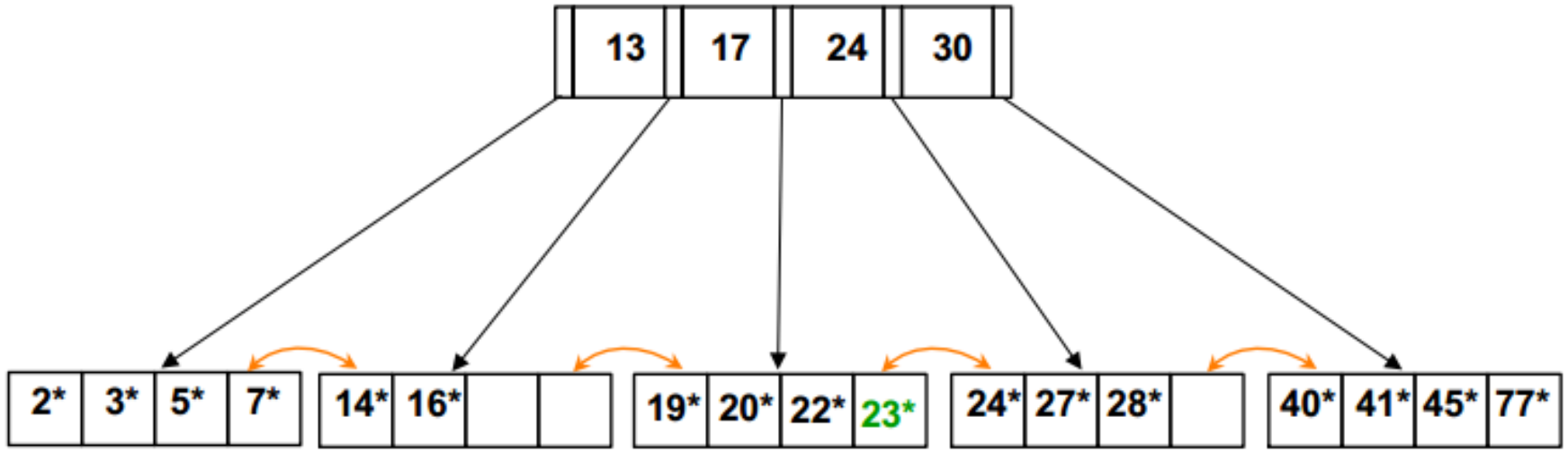
Note: this can happen recursively!

To split the index node, redistribute entries evenly, but push up the middle key. Splitting "grows" a tree; root split increases height.

Example: Insert into B+ Tree

Example: to insert entry 23*

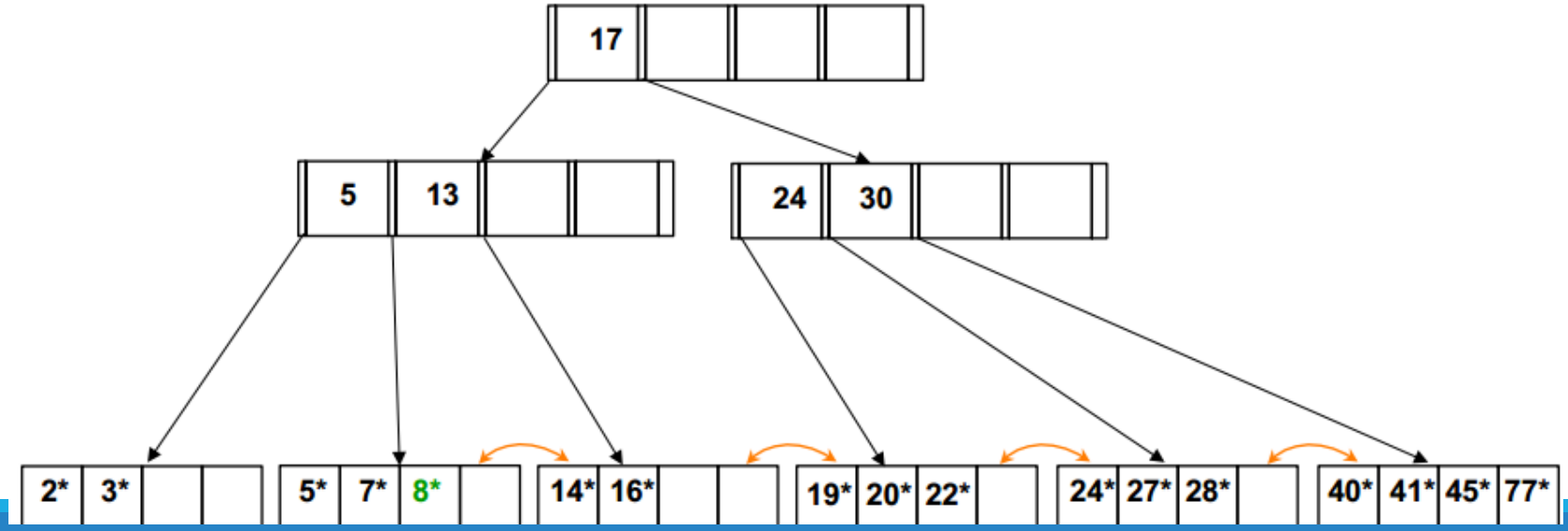
- Follow the third pointer, since 17 ≤ 23 < 24.
- Have enough space, done.



Example: Insert into B+ Tree

Example: to insert entry 8*

- Follow the left-most pointer, since 8 < 13
- Insertion causes overfill; split leaf node into two nodes and redistribute the data evenly.
- “5” is middle key, so it is copied up.
- When inserting “5” into the parent node, it will cause overfill again. “17” is middle key and is pushed up.



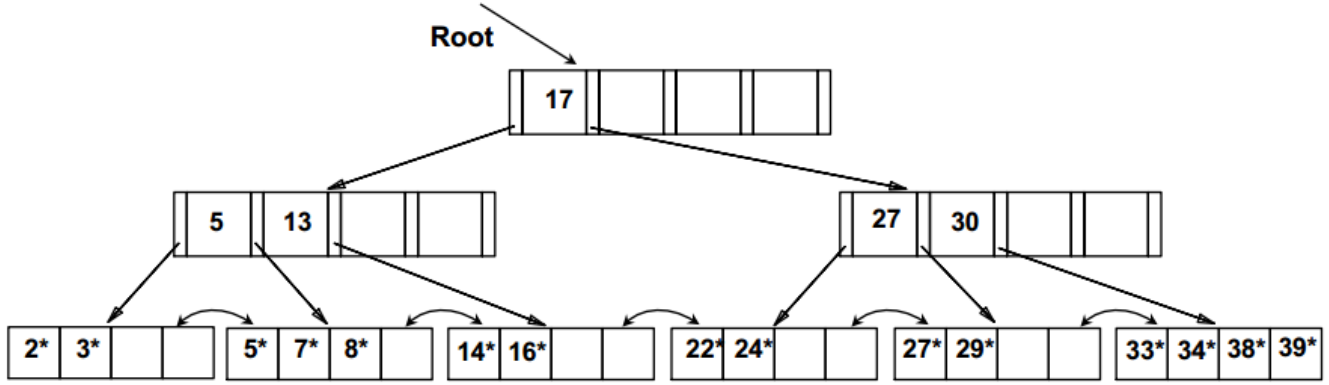
B+ Tree: Deleting a Data Entry

1. Begin at root, find the leaf *L* where entry belongs.
2. Remove the entry.
    i. If *L* is at least half-full, done!
    ii. Else:
        i. Must re-distribute, borrowing from sibling (i.e. adjacent node with same parent as *L*).
        ii. If re-distribution fails, merge *L* and sibling.

Note: merge could propagate to the root!

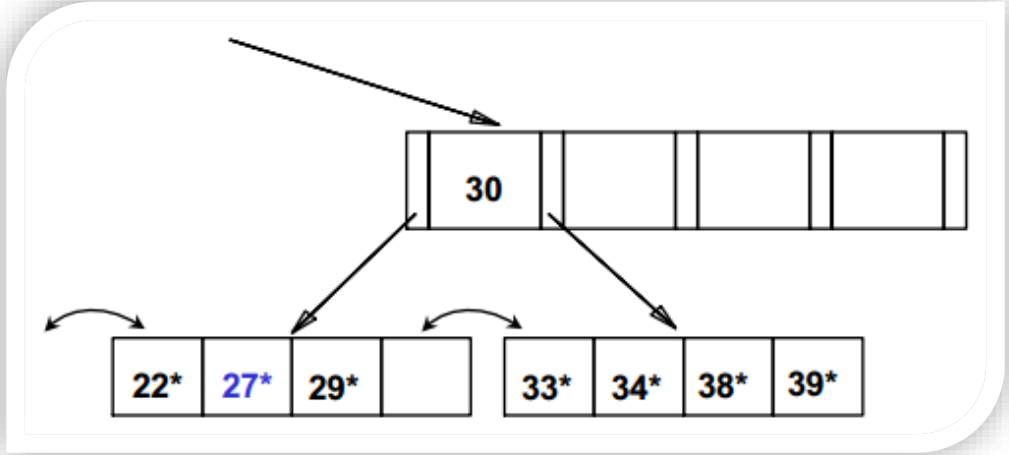
If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.

Example: Deleting from B+ Tree

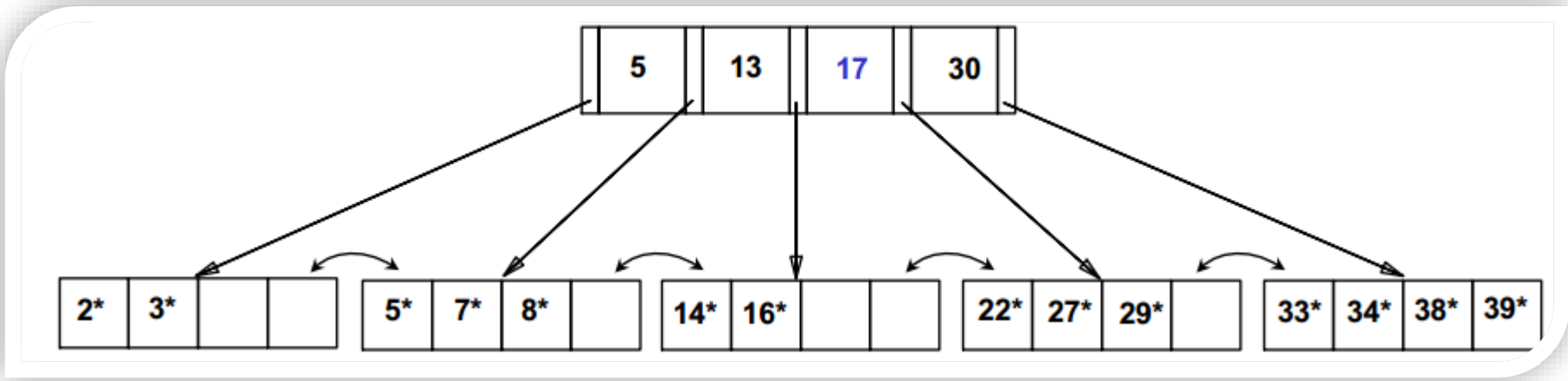


Example: to delete entry 24*

- After the deletion, the leaf contains only one entry 22*, and the sibling contains just two entries. Therefore, we can not redistribute entries.
- Merge leaf and sibling. 'Toss' the entry '27' in the parent.
- Pull down if index entry



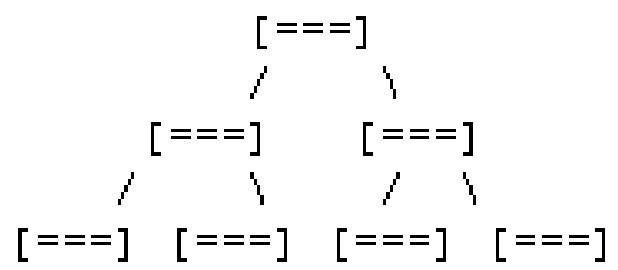
Example: Deleting from B+ Tree



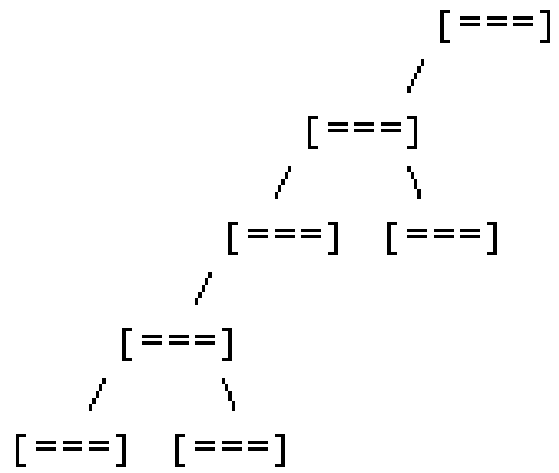
Balanced vs. Unbalanced Trees

In a balanced tree, every path from the route to a leaf node is the same length.

- Balanced



- Unbalanced



UNIVERSITY OF TORONTO MISSISSAUGA

Prefix Key Compression

Key values in index entries only “direct traffic”; more often than not, they can be compressed.

e.g., if we have adjacent index entries with search key values *Dannon Yogurt*, *David Smith*, and *Devarakonda Murthy*, we can abbreviate *David Smith* to *Dav.* The other keys can also be compressed.

Hash-based Indexes

UNIVERSITY OF TORONTO MISSISSAUGA

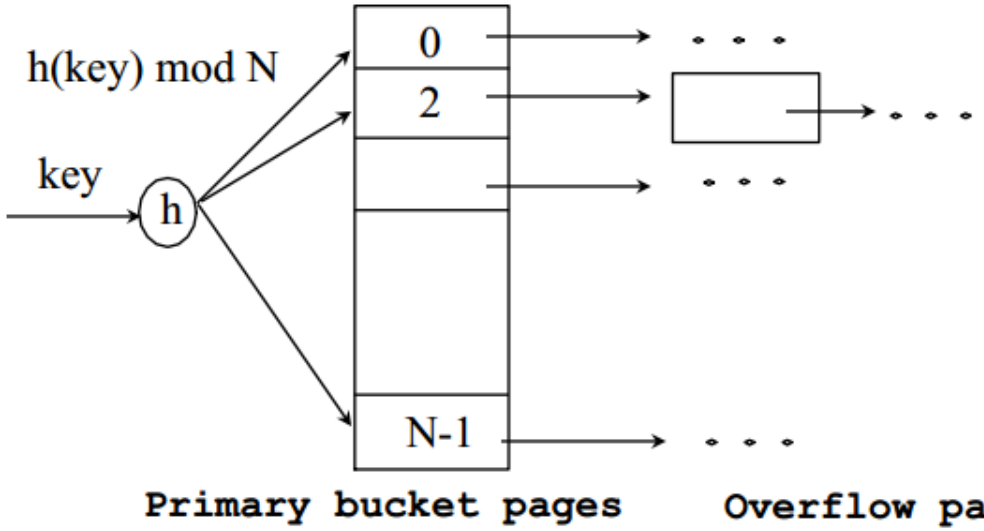
Fast and efficient for equality selections! The Index is a collection of buckets. A bucket is the *primary page* plus zero or more *overflow pages*, they also contain data entries.

Hashing Function ***h***

h(*r*) = bucket in which record *r* belongs. ***h*** looks at the *search key* field of *r*. There is no need for "index entries" in this scheme.

Hash-based Indexes

Hash-based indexes are best for *equality selections*. They do not support efficient range searches.

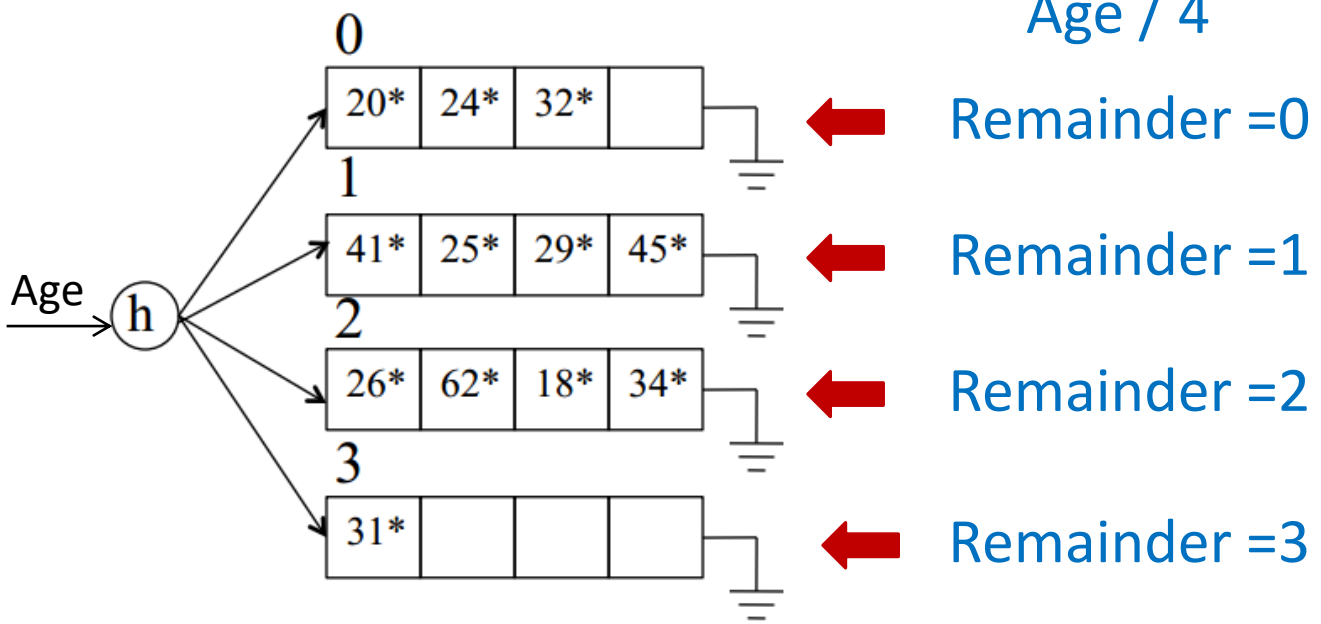


- a collection of buckets 0 through N-1, with one primary page per bucket initially
- bucket = **primary page**+ zero/more **overflow pages**
- buckets contains **data entries**

To search for a data entry, we apply a *hash function* ***h*** to identify the bucket to which it belongs and then search this bucket.

Example: Hash-based Indexes

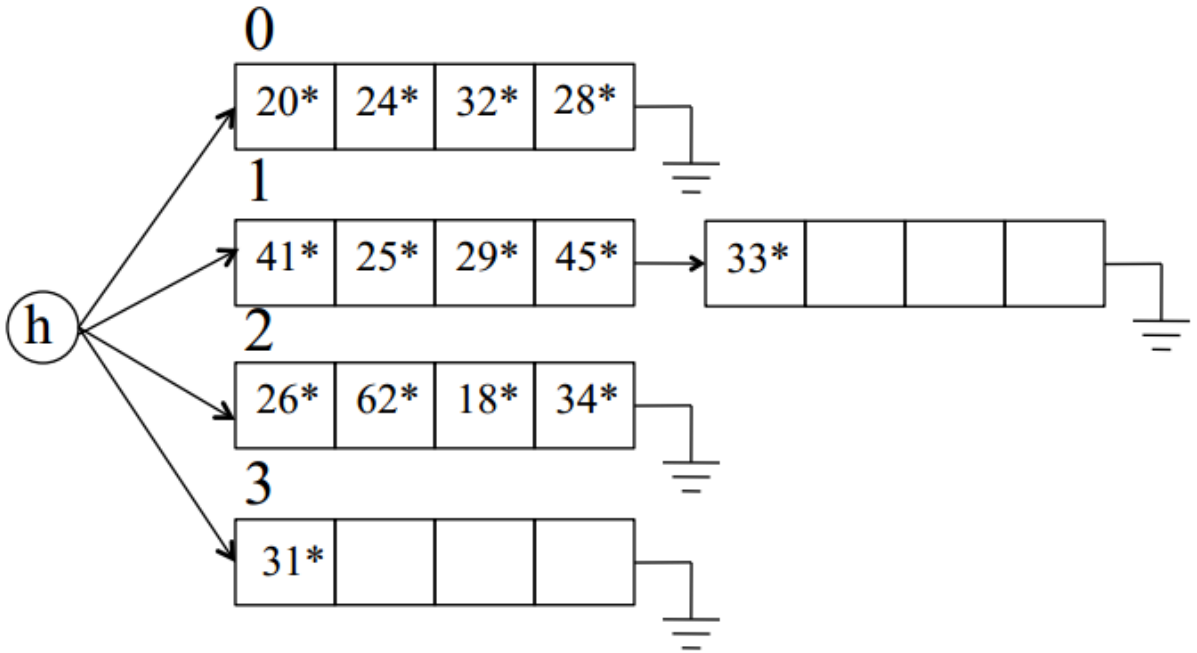
Initially constructed over the "Age" attribute, with 4 records/page and *h*(Age) = Age mod 4.



Example: Hash-based Indexes

To insert a data entry, we use the hash function to identify the correct bucket and then put the data entry there. If there is no space for this data entry, we allocate a new overflow page, put the data entry on this page, and add the page to the overflov

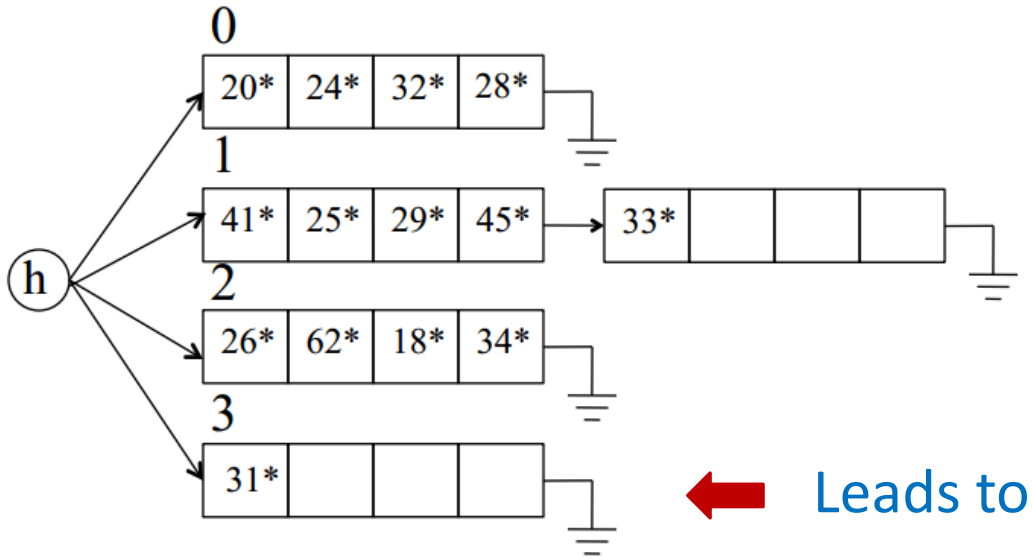
Example: adding 28, 33



Example: Hash-based Indexes

To delete a data entry, we use the hashing function to identify the correct bucket, locate the data entry by searching the bucket, and then remove it. If this data entry is the last in an overflow page, the overflow page is removed from the overflow chain of the bucket and added to a list of free pages.

Example : deleting 31



Leads to empty page

Index Classification

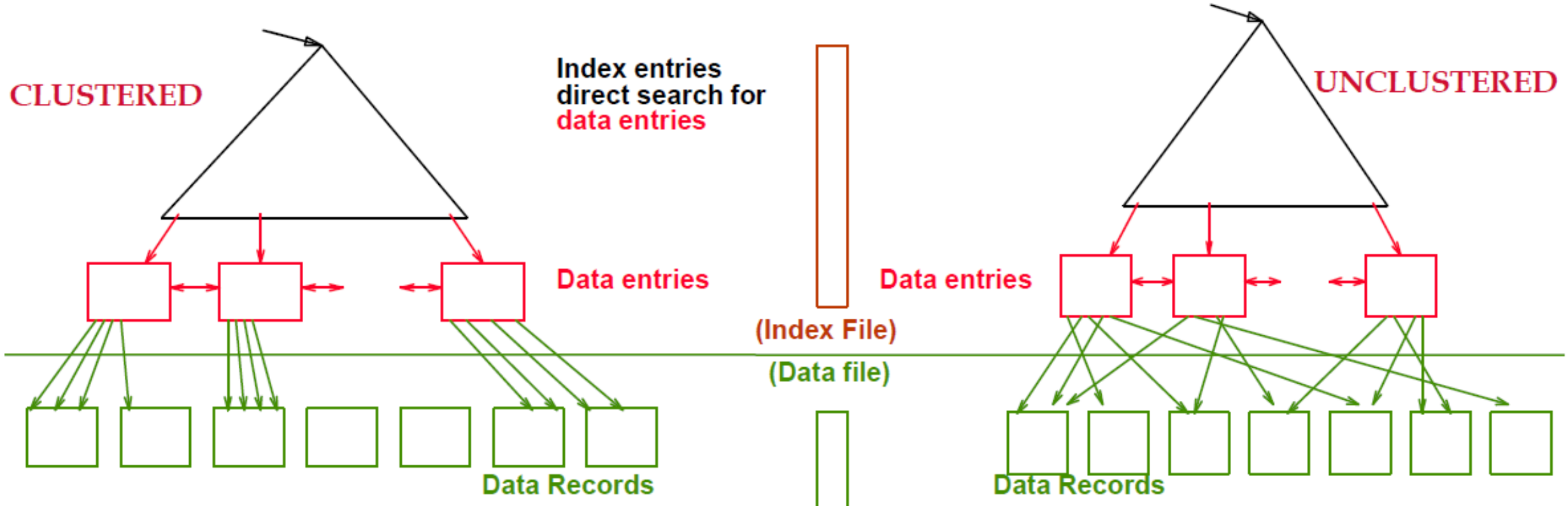
Primary Index vs. Secondary Index

- If a search key contains a primary key, then it is called primary.
  - A Unique Index is one where the search key contains a candidate key.

Clustered Index vs. Unclustered Index

- If the order of index data entries are in the same order as the data records, then it is a clustered index.
  - A file can be clustered on at most one search key.

Index Classification



Understanding the Workload

For each query in the workload:

- Which relations does it access?
- Which attributes are retrieved?
- Which attributed are involved in selection/join conditions?
  - How selective are these conditions likely to be?

For each update in the workload:

- What is the type of modification (INSERT/DELETE/UPDATE)?
- What are the attributes affected?

Choice of Indexes

UNIVERSITY OF TORONTO MISSISSAUGA

What indexes should we create?

- Which relations should have indexes?
- What field(s) should be the search key?
- Should we construct several indexes?

For each index, what kind of an index should it be?

- Clustered? Unclustered?
- Hash? Tree?

Choice of Indexes

A tactic, often used, is to consider the most important queries in order. Consider the best plan using the current indexes and see if a better plan is possible with an additional index.

- This implies an understanding of how a DBMS evaluates queries and creates query evaluation plans.

Before creating a new index, consider the impact on updates in the workload! There is often a trade-off; indexes can make queries go faster but make updates slower. Additionally, they require disk space!

Guidelines

- Attributes in the WHERE clause are candidates for index keys.
  - Exact match conditions suggest the use of a hash index.
  - Range type queries suggest the use of a tree index.
- Multi-Attribute search keys should be considered when a WHERE clause contains several conditions.
- Choose indexes that benefit as many queries as possible.
  - Only one index can be clustered per relation, select it based on important queries that would benefit the most from clustering.

Examples

B+ tree index on E.age can be used to get qualifying tuples.

- Is the index clustered?

```
SELECT  E.dno
FROM  Emp E
WHERE  E.age>40
```

Equality queries and duplicates:

- Indexing on *E.hobby* helps

```
SELECT  E.dno
FROM  Emp E
WHERE  E.hobby=Stamps
```

Composite Search Keys

Composite Search Keys:

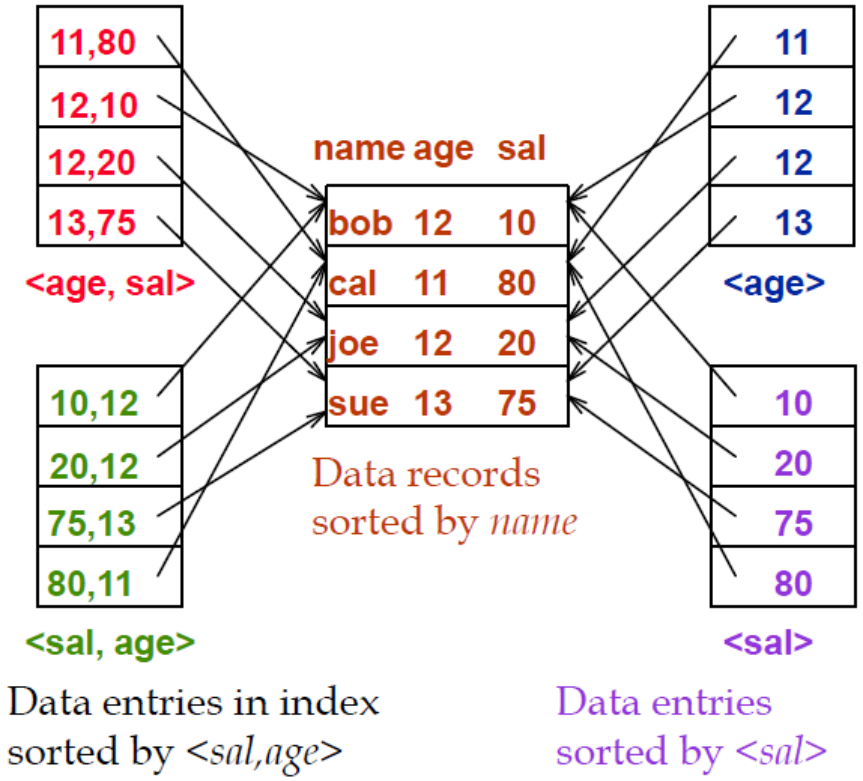
Search on a combination of fields.

- Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
  - age=20 and sal =75
- Range query:
  - age=20 and sal > 10

Data entries in index sorted by search key to support range queries.

- Lexicographic order

Examples of composite key indexes using lexicographic order.



Summary

Many alternative file organizations exist, each appropriate in some situation, and some ‘better’ than others.

If selection queries are frequent, sorting the file or building an *index* is important! REMEMBER:

- Hash-based Indexes are only good for equality searches.
- Tree-based Indexes are best for range searches.

Summary

Can have several indexes on a given file of data records, each with a different search key.

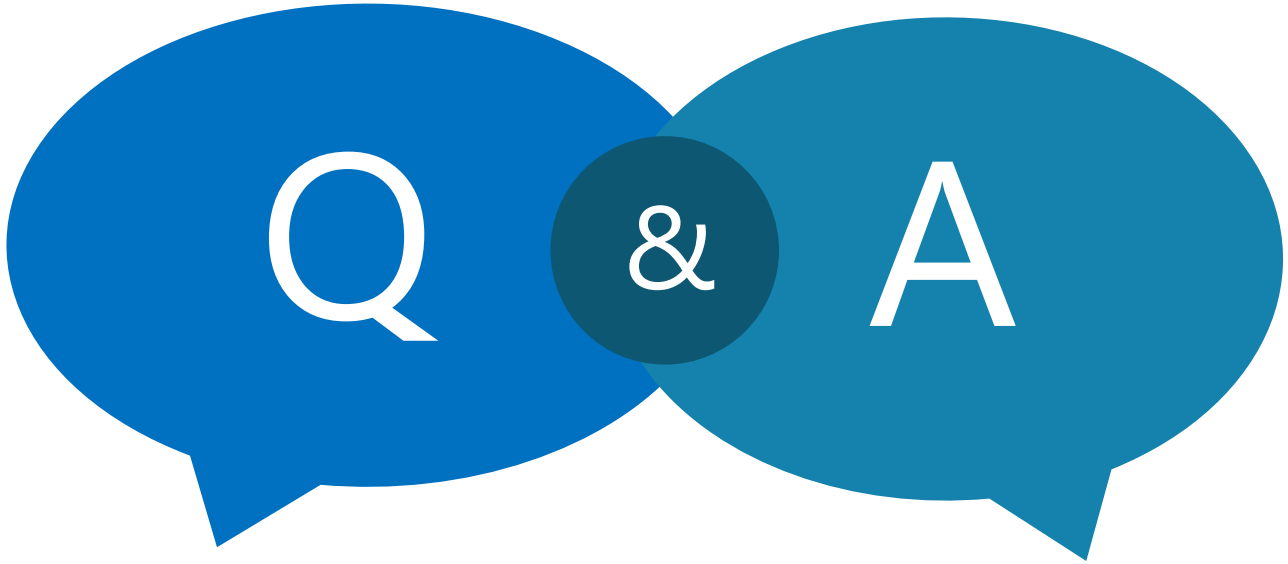
Understanding the nature of the *workload* for the application

- What are the important queries and updates?
- What attributes/relations are involved?

Indexes must be chosen to speed up important queries (and perhaps some updates!).

- Index maintenance overhead on updates to key fields.
- Choose indexes that can help many queries, if/when possible.

Questions?



THANKS FOR LISTENING

I'LL BE ANSWERING QUESTIONS NOW

Citations, Images and Resources

UNIVERSITY OF TORONTO MISSISSAUGA

Database Management Systems (3rd Ed.), Ramakrishnan & Gehrke

Some content is based off the slides of Dr. Fei Chiang - http://www.cas.mcmaster.ca/~fchiang/

Some content is based off the tutorial slides of XiaoJiao Wang - http://www.cas.mcmaster.ca/~wangxj2/personal/SE4DB3_2015/