SQL:
DATA DEFINITION LANGUAGE

Database Schemas in SQL

- SQL is primarily a query language, for getting information from a database.
  - **Data manipulation language (DML)**
- But SQL also includes a *data-definition* component for describing database schemas.
  - **Data definition language (DDL)**

Creating (Declaring) a Relation

- Simplest form is:

```
CREATE TABLE <name> (
        <list of elements>
);
```

- To delete a relation:

```
DROP TABLE <name>;
```

Elements of Table Declarations

- Most basic element: an attribute and its type.
- The most common types are:
  - INT or INTEGER (synonyms).
  - REAL or FLOAT (synonyms).
  - CHAR(*n*) = fixed-length string of *n* characters.
  - VARCHAR(*n*) = variable-length string of up to *n* characters.

Example: Create Table


```
CREATE TABLE Sells (
    bar     CHAR(20),
    beer    VARCHAR(20),
    price   REAL
);
```

SQL Values

- Integers and reals are represented as you would expect.
- Strings are too, except they require single quotes.
  - Two single quotes = real quote, e.g., `'Joe''s Bar'`.
- Any value can be NULL
  - Unless attribute has NOT NULL constraint
  - E.g., price REAL not null,

Dates and Times

- DATE and TIME are types in SQL.
- The form of a date value is:

  DATE 'yyyy-mm-dd'

  - Example: `DATE '2007-09-30'` for Sept. 30, 2007.

Times as Values

- The form of a time value is:

  TIME 'hh:mm:ss'

with an optional decimal point and fractions of a second following.

  - Example: `TIME '15:30:02.5'` = two and a half seconds after 3:30PM.

Declaring Keys

- An attribute or list of attributes may be declared PRIMARY KEY or UNIQUE.
- Either says that no two tuples of the relation may agree in all the attribute(s) on the list.

Our Running Example

Beers(<u>name</u>, manf)

Bars(<u>name</u>, addr, license)

Drinkers(<u>name</u>, addr, phone)

Likes(<u>drinker</u>, <u>beer</u>)

Sells(<u>bar</u>, <u>beer</u>, price)

Frequents(<u>drinker</u>, <u>bar</u>)

- Underline = *key* (tuples cannot have the same value in all key attributes).

Declaring Single-Attribute Keys

- Place PRIMARY KEY or UNIQUE after the type in the declaration of the attribute.
- Example:

```
CREATE TABLE Beers (
    name   CHAR(20) UNIQUE,
    manf   CHAR(20)
);
```

Declaring Multiattribute Keys

- A key declaration can also be another element in the list of elements of a CREATE TABLE statement.
- This form is essential if the key consists of more than one attribute.
  - May be used even for one-attribute keys.

Example: Multiattribute Key

- The bar and beer together are the key for Sells:

```
CREATE TABLE Sells (
    bar        CHAR(20),
    beer       VARCHAR(20),
    price      REAL,
    PRIMARY KEY (bar, beer)
);
```

PRIMARY KEY vs. UNIQUE

1. There can be only one PRIMARY KEY for a relation, but several UNIQUE attributes.
2. No attribute of a PRIMARY KEY can ever be NULL in any tuple. But attributes declared UNIQUE may have NULL's, and there may be several tuples with NULL.

Kinds of Constraints

- Keys
- Foreign-key, or referential-integrity.
- Domain constraints
  - Constrain values of a particular attribute.
- Tuple-based constraints
  - Relationship among components.
- Assertions: any SQL boolean expression

Foreign Keys

- Values appearing in attributes of one relation must appear together in certain attributes of another relation.
- Example: in Sells(bar, beer, price), we might expect that a beer value also appears in Beers.name

Expressing Foreign Keys

- Use keyword REFERENCES, either:
  1. After an attribute (for one-attribute keys).
  2. As an element of the schema:

  FOREIGN KEY (<list of attributes>)

  REFERENCES <relation> (<attributes>)
- Referenced attributes must be declared PRIMARY KEY or UNIQUE.

Example: With Attribute


```
CREATE TABLE Beers (
  name      CHAR(20) PRIMARY KEY,
  manf      CHAR(20) );

CREATE TABLE Sells (
  bar       CHAR(20),
  beer      CHAR(20) REFERENCES Beers(name),
  price     REAL );
```

Example: As Schema Element


```
CREATE TABLE Beers (
  name      CHAR(20) PRIMARY KEY,
  manf      CHAR(20) );

CREATE TABLE Sells (
  bar       CHAR(20),
  beer      CHAR(20),
  price     REAL,
  FOREIGN KEY(beer) REFERENCES
      Beers(name));
```

Enforcing Foreign-Key Constraints

- If there is a foreign-key constraint from relation *R* to relation *S*, two violations are possible:
  1. An insert or update to *R* introduces values not found in *S*.
  2. A deletion or update to S causes some tuples of *R* to "dangle."

Actions Taken --- (1)

- Example: suppose R = Sells, S = Beers.
- An insert or update to Sells that introduces a nonexistent beer must be rejected.
- A deletion or update to Beers that removes a beer value found in some tuples of Sells can be handled in three ways…

Actions Taken --- (2)

1. *Default* : Reject the modification.
2. *Cascade* : Make the same changes in Sells.
   - Deleted beer: delete Sells tuple.
   - Updated beer: change value in Sells.
3. *Set NULL* : Change the beer to NULL.

Example: Cascade

- Delete the Bud tuple from Beers:
  - Then delete all tuples from Sells that have beer = ’Bud’.
- Update the Bud tuple by changing ’Bud’ to ’Budweiser’:
  - Then change all Sells tuples with beer = ’Bud’ to beer = ’Budweiser’.

Example: Set NULL

- Delete the Bud tuple from Beers:
  - Change all tuples of Sells that have beer = ’Bud’ to have beer = NULL.
- Update the Bud tuple by changing ’Bud’ to ’Budweiser’:
  - Same change as for deletion.

Choosing a Policy

- When we declare a foreign key, we may choose policies SET NULL or CASCADE independently for deletions and updates.
- Follow the foreign-key declaration by:

ON [UPDATE, DELETE][SET NULL CASCADE]

- Two such clauses may be used.
- Otherwise, the default (reject) is used.

Example: Setting Policy


```
CREATE TABLE Sells (
  bar     CHAR(20),
  beer    CHAR(20),
  price   REAL,
  FOREIGN KEY(beer)
     REFERENCES Beers(name)
     ON DELETE SET NULL
     ON UPDATE CASCADE
);
```

Attribute-Based Checks

- Constraints on the value of a particular attribute.
- Add CHECK(<condition>) to the declaration for the attribute.
- The condition may use the name of the attribute, but any other relation or attribute name must be in a subquery.

Example: Attribute-Based Check


```
CREATE TABLE Sells (
  bar     CHAR(20),
  beer    CHAR(20) CHECK ( beer IN
          (SELECT name FROM Beers)),
  price   REAL CHECK ( price <= 5.00 )
);
```

Timing of Checks

- Attribute-based checks are performed only when a value for that attribute is inserted or updated.
  - Example: `CHECK (price <= 5.00)` checks every new price and rejects the modification (for that tuple) if the price is more than $5.
  - Example: `CHECK (beer IN (SELECT name FROM Beers))` not checked if a beer is deleted from Beers (unlike foreign-keys).

Tuple-Based Checks

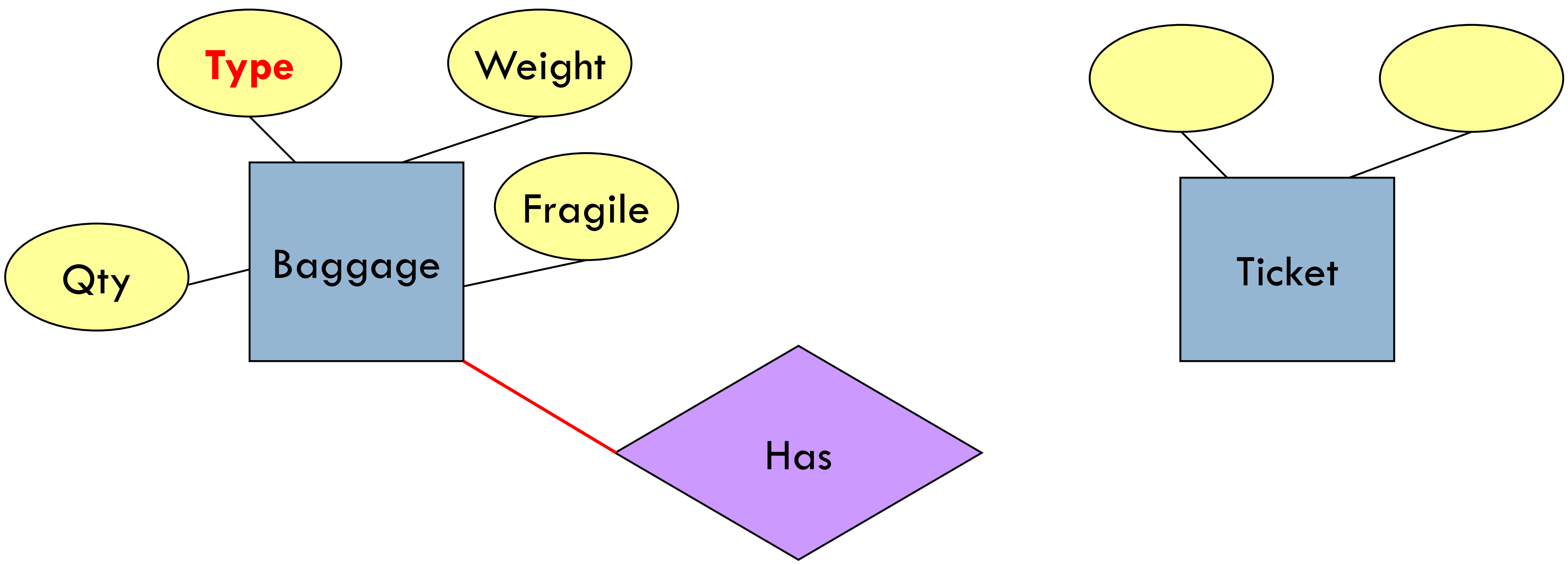
- CHECK (<condition>) may be added as a relation-schema element.
- The condition may refer to any attribute of the relation.
  - But other attributes or relations require a subquery.
- Checked on insert or update only.

Example: Tuple-Based Check

- Only Joe’s Bar can sell beer for more than $5:

```
CREATE TABLE Sells (
    bar         CHAR(20),
    beer        CHAR(20),
    price       REAL,
    CHECK (bar = 'Joe''s Bar' OR
                 price <= 5.00)
);
```

Asg 1 Update: Missing attribute in Baggage



INTRODUCTION TO SQL

Why SQL?

- SQL is a very-high-level language.
  - Structured Query Language
  - Say "what to do" rather than "how to do it."
  - Avoid a lot of data-manipulation details needed in procedural languages like C++ or Java.
- Database management system figures out "best" way to execute query.
  - Called "query optimization."

Credit: Renee J. Miller

Database Schemas in SQL

- SQL is primarily a query language, for getting information from a database.
  - **Data manipulation language (DML)**
- But SQL also includes a *data-definition* component for describing database schemas.
  - **Data definition language (DDL)**

Select-From-Where Statements

SELECT desired attributes

FROM one or more tables

WHERE condition about tuples of

the tables

Our Running Example

- Our SQL queries will be based on the following database schema.
  - Underline indicates key attributes.

Beers(<u>name</u>, manf)

Bars(<u>name</u>, addr, license)

Drinkers(<u>name</u>, addr, phone)

Likes(<u>drinker</u>, <u>beer</u>)

Sells(<u>bar</u>, <u>beer</u>, price)

Frequents(<u>drinker</u>, <u>bar</u>)

Example

- Using Beers(name, manf), what beers are made by Anheuser-Busch?

```
SELECT name
FROM Beers
WHERE manf = 'Anheuser-Busch';
```

Result of Query

name
Bud
Bud Lite
Michelob
. . .

The answer is a relation with a single attribute, name, and tuples with the name of each beer by Anheuser-Busch, such as Bud.

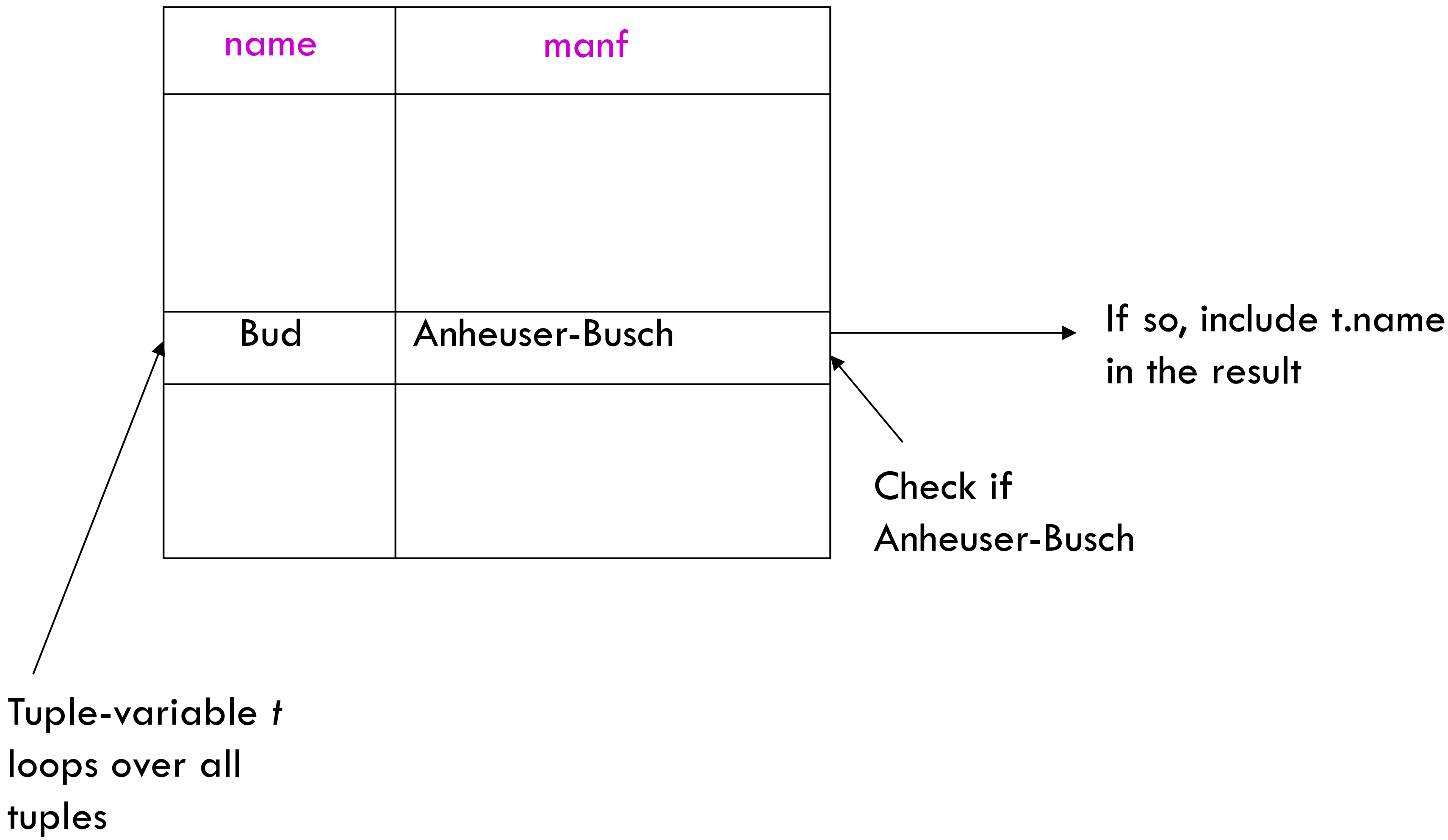
Meaning of Single-Relation Query

- Begin with the relation in the FROM clause.
- Apply the selection indicated by the WHERE clause.
- Apply the extended projection indicated by the SELECT clause.

Operational Semantics - General

- Think of a *tuple variable* visiting each tuple of the relation mentioned in FROM.
- Check if the tuple assigned to the tuple variable satisfies the WHERE clause.
- If so, compute the attributes or expressions of the SELECT clause using the components of this tuple.

Operational Semantics



Example

- What beers are made by Anheuser-Busch?

```
SELECT name
FROM Beers
WHERE manf = 'Anheuser-Busch';
```

OR:

```
SELECT t.name
FROM Beers t
WHERE t.manf ='Anheuser-Busch';
```

Note: these are identical queries.

* In SELECT clauses

- When there is one relation in the FROM clause, * in the SELECT clause stands for "all attributes of this relation."
- Example: Using Beers(name, manf):

```
SELECT *
FROM Beers
WHERE manf = 'Anheuser-Busch';
```

Result of Query:

name	manf
Bud	Anheuser-Busch
Bud Lite	Anheuser-Busch
Michelob	Anheuser-Busch
. . .	. . .

Now, the result has each of the attributes of Beers.

Renaming Attributes

- If you want the result to have different attribute names, use "AS <new name>" to rename an attribute.
- Example: Using Beers(name, manf):

```
SELECT name AS beer, manf
FROM Beers
WHERE manf = 'Anheuser-Busch'
```

Result of Query:

beer	manf
Bud	Anheuser-Busch
Bud Lite	Anheuser-Busch
Michelob	Anheuser-Busch
. . .	. . .

Expressions in SELECT Clauses

- Any valid expression can appear as an element of a SELECT clause.
- Example: Using Sells(bar, beer, price):

```
SELECT bar, beer,
       price*95 AS priceInYen
FROM Sells;
```

Result of Query

bar	beer	priceInYen
Joe's	Bud	285
Sue's	Miller	342
...	...	...

Example: Constants as Expressions

- Using Likes(drinker, beer):

```
SELECT drinker,
       'likes Bud' AS whoLikesBud
FROM Likes
WHERE beer = 'Bud';
```

Result of Query

drinker	whoLikesBud
Sally	likes Bud
Fred	likes Bud
. . .	. . .

Complex Conditions in WHERE Clause

- Boolean operators AND, OR, NOT.
- Comparisons =, <>, <, >, <=, >=.

Example: Complex Condition

- Using Sells(bar, beer, price), find the price Joe's Bar charges for Bud:

```
SELECT  price
FROM    Sells
WHERE bar = 'Joe''s Bar' AND
      beer = 'Bud';
```

Patterns

- A condition can compare a string to a pattern by:
  - <Attribute> LIKE <pattern> or <Attribute> NOT LIKE <pattern>
- *Pattern* is a quoted string
  - % = "any string";
  - _ = "any character".

Example: LIKE

- Using Drinkers(name, addr, phone) find the drinkers with exchange 555:

```
SELECT name
FROM Drinkers
WHERE phone LIKE '%555-_ _ _ _';
```

NULL Values

- Tuples in SQL relations can have NULL as a value for one or more components.
- Meaning depends on context. Two common cases:
  - *Missing value* : e.g., we know Joe's Bar has some address, but we don't know what it is.
  - *Inapplicable* : e.g., the value of attribute spouse for an unmarried person.

Comparing NULL's to Values

- The logic of conditions in SQL is really 3-valued logic: TRUE, FALSE, UNKNOWN.
- Comparing any value (including NULL itself) with NULL yields UNKNOWN.
- A tuple is in a query answer iff the WHERE clause is TRUE (not FALSE or UNKNOWN).

Three-Valued Logic

- To understand how AND, OR, and NOT work in 3-valued logic
- For TRUE result
  - OR: at least one operand must be TRUE
  - AND: both operands must be TRUE
  - NOT: operand must be FALSE
- For FALSE result
  - OR: both operands must be FALSE
  - AND: at least one operand must be FALSE
  - NOT: operand must be TRUE
- Otherwise, result is UNKNOWN

Example

- From the following Sells relation:

bar	beer	price
Joe's Bar	Bud	NULL

SELECT bar

FROM Sells

WHERE price < 2.00 OR price >= 5.00;

UNKNOWN   UNKNOWN

UNKNOWN

Multi-Relation Queries

- Interesting queries often combine data from more than one relation.
- We can address several relations in one query by listing them all in the FROM clause.
- Distinguish attributes of the same name by "<relation>.<attribute>" .

Example: Joining Two Relations

- Using relations Likes(drinker, beer) and Frequents(drinker, bar), find the beers liked by at least one person who frequents Joe's Bar.

```
SELECT beer
FROM Likes, Frequents
WHERE bar = 'Joe''s Bar' AND
      Frequents.drinker = Likes.drinker;
```

Example: Joining Two Relations

- Alternatively can use explicit (named) tuple variables

```
SELECT beer
 FROM Likes l, Frequents f
 WHERE bar = ’Joe’’s Bar’ AND
    f.drinker = l.drinker;
```

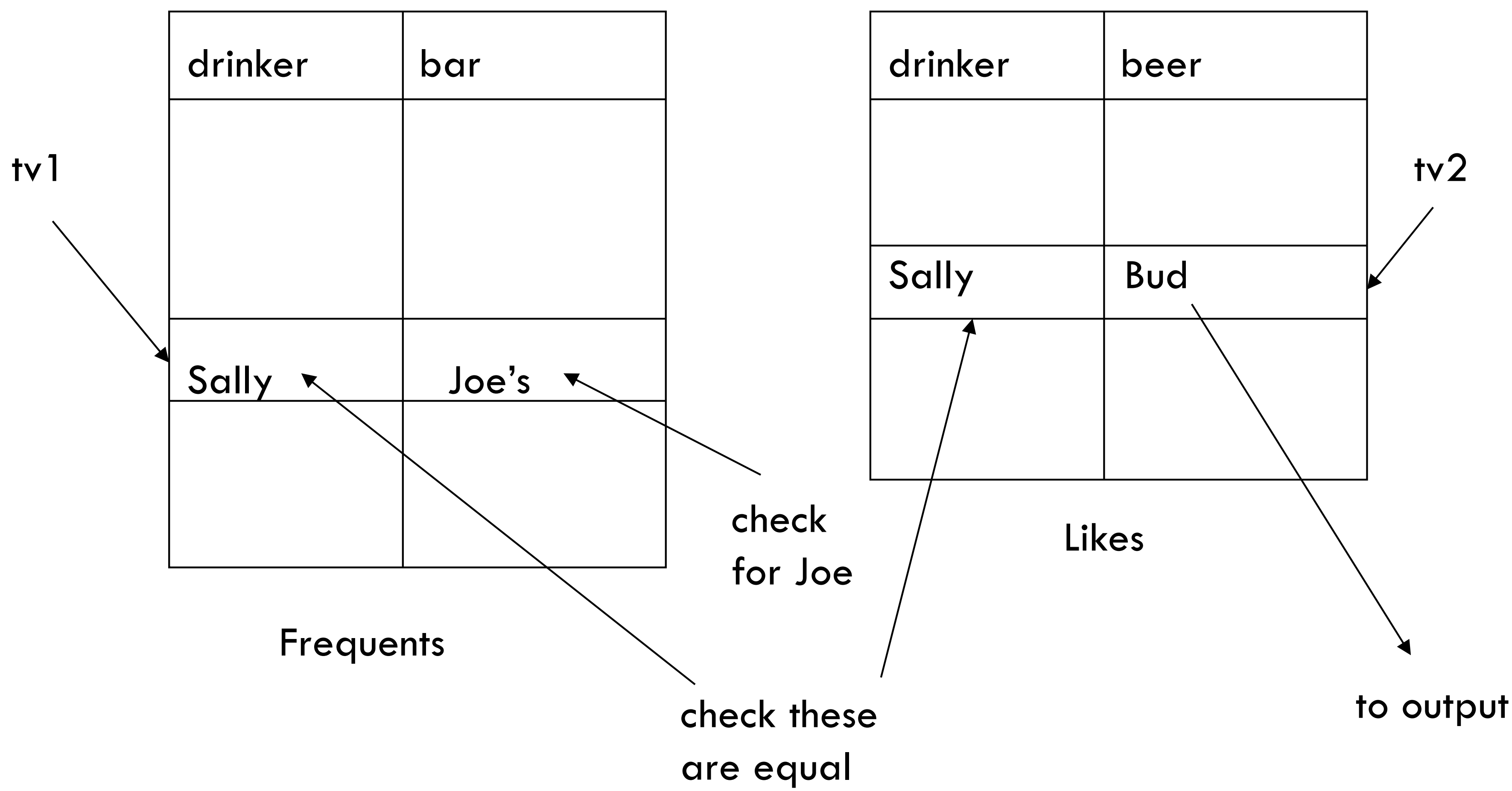
Formal Semantics

- Almost the same as for single-relation queries:
  - Start with the product of all the relations in the FROM clause.
  - Apply the selection condition from the WHERE clause.
  - Project onto the list of attributes and expressions in the SELECT clause.

Operational Semantics

- Imagine one tuple-variable for each relation in the FROM clause.
  - These tuple-variables visit each combination of tuples, one from each relation.
- If the tuple-variables are pointing to tuples that satisfy the WHERE clause, send these tuples to the SELECT clause.

Example


drinker
bar
tv1
Sally
Joe's
Frequents
check
for Joe
drinker
beer
tv2
Sally
Bud
Likes
check these
are equal
to output

Explicit Tuple-Variables

- Sometimes, a query needs to use two copies of the same relation.
- Distinguish copies by following the relation name by the name of a tuple-variable, in the FROM clause.
- It’s always an option to rename relations this way, even when not essential.

Example: Self-Join

- From Beers(name, manf), find all pairs of beers by the same manufacturer.
  - Do not produce pairs like (Bud, Bud).
  - Do not produce the same pair twice like (Bud, Miller) and (Miller, Bud).

Select-From-Where Statements

SELECT desired attributes

FROM one or more tables

WHERE condition about tuples of

the tables

Example

- Using Beers(name, manf), what beers are made by Anheuser-Busch?

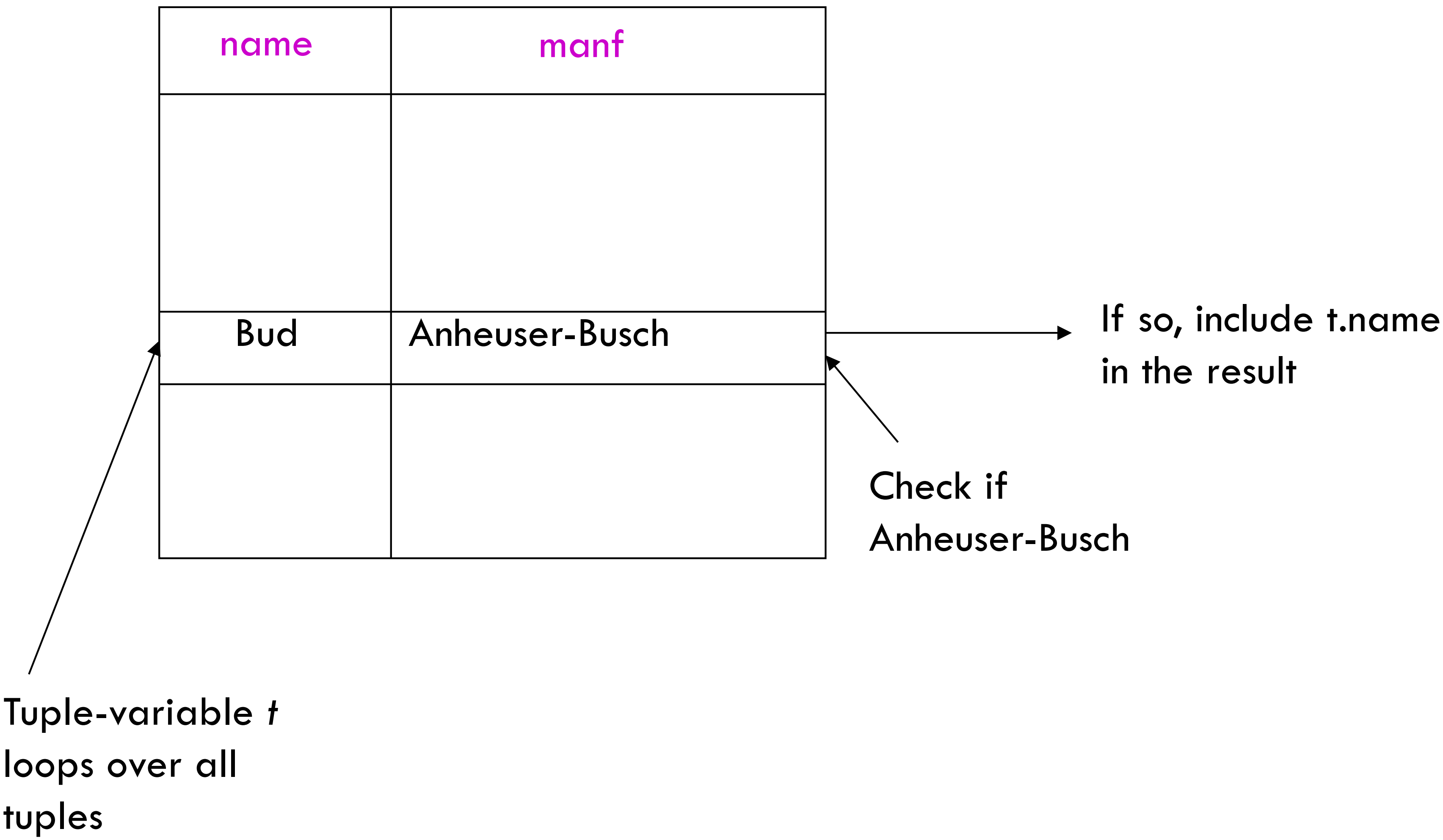
```
SELECT name
FROM Beers
WHERE manf = 'Anheuser-Busch';
```

Result of Query

name
Bud
Bud Lite
Michelob
. . .

The answer is a relation with a single attribute, name, and tuples with the name of each beer by Anheuser-Busch, such as Bud.

Operational Semantics



Comparing NULL's to Values

- The logic of conditions in SQL is really 3-valued logic: TRUE, FALSE, UNKNOWN.
- Comparing any value (including NULL itself) with NULL yields UNKNOWN.
- A tuple is in a query answer iff the WHERE clause is TRUE (not FALSE or UNKNOWN).

Three-Valued Logic

- To understand how AND, OR, and NOT work in 3-valued logic
- For TRUE result
  - OR: at least one operand must be TRUE
  - AND: both operands must be TRUE
  - NOT: operand must be FALSE
- For FALSE result
  - OR: both operands must be FALSE
  - AND: at least one operand must be FALSE
  - NOT: operand must be TRUE
- Otherwise, result is UNKNOWN

Example

- From the following Sells relation:

bar	beer	price
Joe's Bar	Bud	NULL

SELECT bar

FROM Sells

WHERE price < 2.00 OR price >= 5.00;

UNKNOWN UNKNOWN

UNKNOWN

Multi-Relation Queries

- Interesting queries often combine data from more than one relation.
- We can address several relations in one query by listing them all in the FROM clause.
- Distinguish attributes of the same name by "<relation>.<attribute>" .

Example: Joining Two Relations

- Using relations Likes(drinker, beer) and Frequents(drinker, bar), find the beers liked by at least one person who frequents Joe's Bar.

```
SELECT beer
FROM Likes, Frequents
WHERE bar = 'Joe''s Bar' AND
      Frequents.drinker = Likes.drinker;
```

Example: Joining Two Relations

- Alternatively can use explicit (named) tuple variables

```
SELECT beer
 FROM Likes l, Frequents f
 WHERE bar = 'Joe''s Bar' AND
    f.drinker = l.drinker;
```

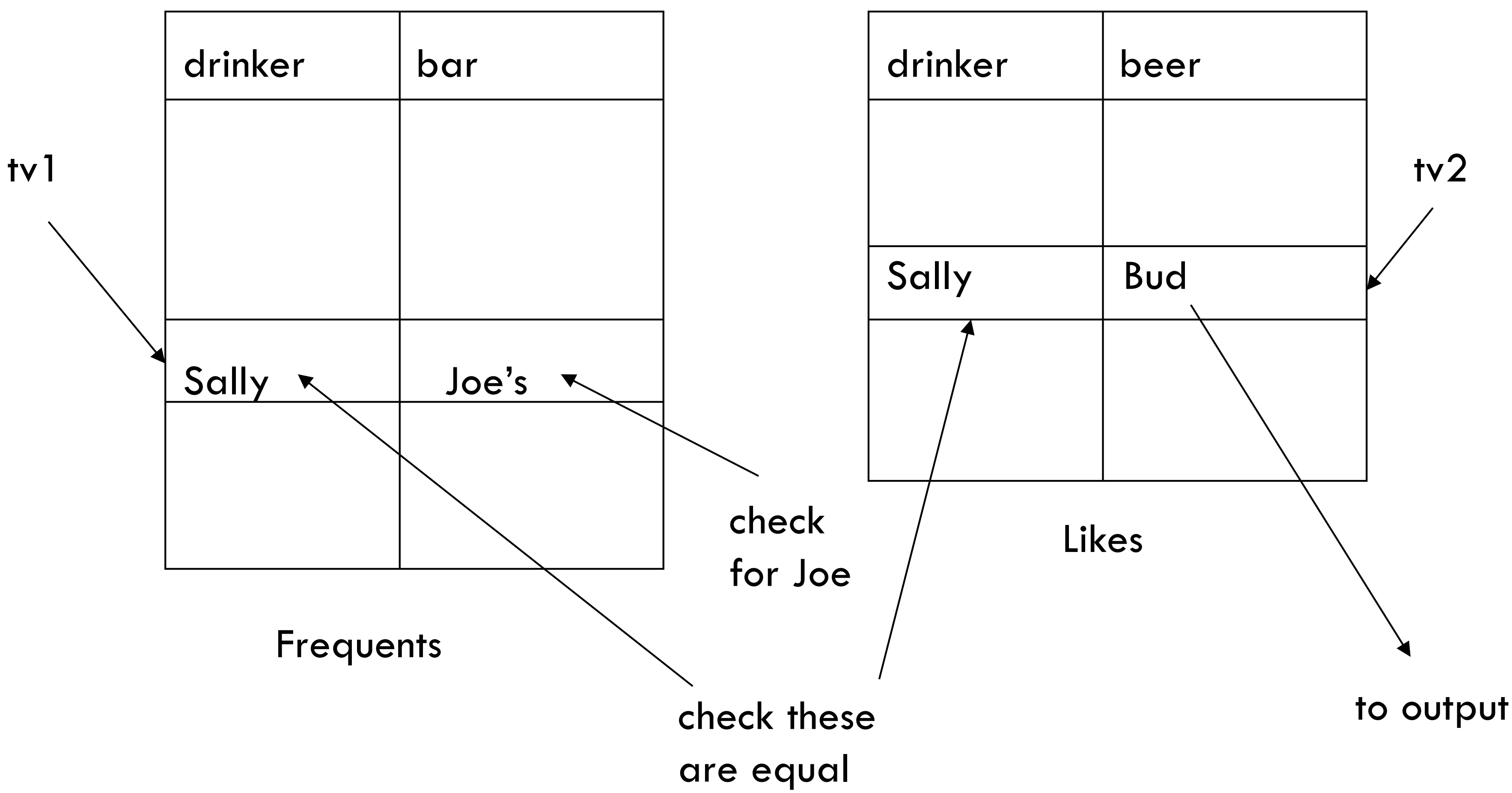
Formal Semantics

- Almost the same as for single-relation queries:
  - Start with the product of all the relations in the FROM clause.
  - Apply the selection condition from the WHERE clause.
  - Project onto the list of attributes and expressions in the SELECT clause.

Operational Semantics

- Imagine one tuple-variable for each relation in the FROM clause.
  - These tuple-variables visit each combination of tuples, one from each relation.
- If the tuple-variables are pointing to tuples that satisfy the WHERE clause, send these tuples to the SELECT clause.

Example



tv1
drinker
bar
Sally
Joe's
Frequents
check
for Joe
check these
are equal
drinker
beer
Sally
Bud
tv2
Likes
to output

Explicit Tuple-Variables

- Sometimes, a query needs to use two copies of the same relation.
- Distinguish copies by following the relation name by the name of a tuple-variable, in the FROM clause.
- It’s always an option to rename relations this way, even when not essential.

Example: Self-Join

- From Beers(name, manf), find all pairs of beers by the same manufacturer.
  - Do not produce pairs like (Bud, Bud).
  - Do not produce the same pair twice like (Bud, Miller) and (Miller, Bud).

```
SELECT b1.name, b2.name
FROM Beers b1, Beers b2
WHERE b1.manf = b2.manf AND
      b1.name < b2.name;
```

Subqueries

- A parenthesized SELECT-FROM-WHERE statement (*subquery*) can be used as a value in a number of places, including FROM and WHERE clauses.
- Example: in place of a relation in the FROM clause, we can use a subquery and then query its result.
  - Must use a tuple-variable to name tuples of the result.

Example: Subquery in FROM

- Find the beers liked by at least one person who frequents Joe's Bar.

Drinkers who frequent Joe's Bar

```
SELECT beer
FROM Likes, (SELECT drinker
             FROM Frequents
         WHERE bar = 'Joe''s Bar')JD
WHERE Likes.drinker = JD.drinker;
```

Subqueries often obscure queries

- Find the beers liked by at least one person who frequents Joe's Bar.

```
SELECT beer
FROM Likes l, Frequents f
WHERE l.drinker = f.drinker AND
   bar = 'Joe''s Bar';
```

Simple join query

Subqueries That Return One Tuple

- If a subquery is guaranteed to produce one tuple, then the subquery can be used as a value.
  - Usually, the tuple has one component.
  - Remember SQL's 3-valued logic.

Example: Single-Tuple Subquery

- Using Sells(bar, beer, price), find the bars that serve Miller for the same price Joe charges for Bud.

Two queries would work:

- Find the price Joe charges for Bud.
- Find the bars that serve Miller at that price.

Query + Subquery Solution

- Find the price Joe charges for Bud.
- Find the bars that serve Miller at that price.

Sells(bar, beer, price)

```
SELECT bar
FROM Sells
WHERE beer = 'Miller' AND price
    = (SELECT price
       FROM Sells
       WHERE bar = 'Joe''s Bar'
         AND beer = 'Bud');
```

The price at which Joe sells Bud

What if price of Bud is NULL?

Query + Subquery Solution


```
SELECT bar
FROM Sells
WHERE beer = 'Miller' AND
    price = (SELECT price
             FROM Sells
             WHERE beer = 'Bud');
```

What if subquery returns multiple values?

Recap: Conditions in WHERE Clause

- Boolean operators AND, OR, NOT.
- Comparisons =, <>, <, >, <=, >=.
- LIKE operator

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, $\geq$ \$90,000 and $\leq$ \$100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000

Subqueries

- A parenthesized SELECT-FROM-WHERE statement (*subquery*) can be used as a value in a number of places, including FROM and WHERE clauses.
- Example: in place of a relation in the FROM clause, we can use a subquery and then query its result.
  - Must use a tuple-variable to name tuples of the result.

Example: Subquery in FROM

- Find the beers liked by at least one person who frequents Joe's Bar.

Drinkers who frequent Joe's Bar

```
SELECT beer
FROM Likes, (SELECT drinker
            FROM Frequents
         WHERE bar = 'Joe''s Bar')JD
WHERE Likes.drinker = JD.drinker;
```

Subqueries often obscure queries

- Find the beers liked by at least one person who frequents Joe's Bar.

```
SELECT beer
FROM Likes l, Frequents f
WHERE l.drinker = f.drinker AND
   bar = 'Joe''s Bar';
```

Simple join query

Subqueries That Return One Tuple

- If a subquery is guaranteed to produce one tuple, then the subquery can be used as a value.
  - Usually, the tuple has one component.
  - Remember SQL's 3-valued logic.

Example: Single-Tuple Subquery

- Using Sells(bar, beer, price), find the bars that serve Miller for the same price Joe charges for Bud.

Two queries would work:

- Find the price Joe charges for Bud.
- Find the bars that serve Miller at that price.

Query + Subquery Solution

- Find the price Joe charges for Bud.
- Find the bars that serve Miller at that price.

Sells(bar, beer, price)

```
SELECT bar
FROM Sells
WHERE beer = 'Miller' AND price
    = (SELECT price
       FROM Sells
       WHERE bar = 'Joe''s Bar'
          AND beer = 'Bud');
```

The price at which Joe sells Bud

What if price of Bud is NULL?

Query + Subquery Solution


```
SELECT bar
FROM Sells
WHERE beer = 'Miller' AND
    price = (SELECT price
             FROM Sells
             WHERE beer = 'Bud');
```

What if subquery returns multiple values?

Recap: Conditions in WHERE Clause

- Boolean operators AND, OR, NOT.
- Comparisons =, <>, <, >, <=, >=.
- LIKE operator

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, $\geq$ \$90,000 and $\leq$ \$100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000

The Operator ANY

- *x* = ANY(<subquery>) is a boolean condition that is true iff *x* equals at least one tuple in the subquery result.
  - = could be any comparison operator.
- Example: *x* >= ANY(<subquery>) means *x* is not the uniquely smallest tuple produced by the subquery.
  - Note tuples must have one component only.

The Operator ALL

- *x* <> ALL(<subquery>) is true iff for every tuple *t* in the relation, *x* is not equal to *t*.
  - That is, *x* is not in the subquery result.
- <> can be any comparison operator.
- Example: *x* >= ALL(<subquery>) means there is no tuple larger than *x* in the subquery result.

Example: ALL

- From Sells(bar, beer, price), find the beer(s) sold for the highest price.

```
SELECT beer
FROM Sells
WHERE price >=
    ALL( SELECT price
         FROM Sells);
```

price from the outer Sells must not be less than any price.

The IN Operator

- <value> IN (<subquery>) is true if and only if the <value> is a member of the relation produced by the subquery.
  - Opposite: <value> NOT IN (<subquery>).
- IN-expressions can appear in WHERE clauses.
- WHERE col IN (value1, value2, ...)

IN is Concise

- SELECT * FROM Cartoons
  WHERE LastName IN (‘Simpsons', 'Smurfs’, ’Flintstones’)

- SELECT * FROM Cartoons

WHERE LastName = ‘Simpsons’

OR LastName = ‘Smurfs’

OR LastName = ‘Flintstones’

Example: IN

- Using Beers(name, manf) and Likes(drinker, beer), find the name and manufacturer of each beer that Fred likes.

```
SELECT *
FROM Beers
WHERE name IN (SELECT beer
               FROM Likes
               WHERE drinker = 'Fred');
```

The set of beers Fred likes

IN vs. Join


```
SELECT R.a
FROM R, S
WHERE R.b = S.b;
```

```
SELECT R.a
FROM R
WHERE b IN (SELECT b FROM S);
```

IN is a Predicate About R's Tuples


```
SELECT a
FROM R
WHERE b IN (SELECT b FROM S);
```

Two 2's

One loop, over the tuples of R

R

a	b
1	2
3	4

S

b	c
2	5
2	6

(1,2) satisfies the condition; 1 is output once.

This Query Pairs Tuples from R, S


```
SELECT a
FROM R, S
WHERE R.b = S.b;
```

Double loop, over the tuples of R and S

a	b
1	2
3	4

R

b	c
2	5
2	6

S

(1,2) with (2,5) and (1,2) with (2,6) both satisfy the condition;
1 is output twice.

Query + Subquery Solution


```
SELECT bar
FROM Sells
WHERE beer = 'Miller' AND
    price = (SELECT price
             FROM Sells
             WHERE beer = 'Bud');
```

What if subquery returns multiple values?

The Operator ANY

- x = ANY(<subquery>) is a boolean condition that is true iff x equals at least one tuple in the subquery result.
  - = could be any comparison operator.
- Example: x >= ANY(<subquery>) means x is not the uniquely smallest tuple produced by the subquery.
  - Note tuples must have one component only.

The Operator ALL

- *x* <> ALL(<subquery>) is true iff for every tuple *t* in the relation, *x* is not equal to *t*.
  - That is, *x* is not in the subquery result.
- <> can be any comparison operator.
- Example: *x* >= ALL(<subquery>) means there is no tuple larger than *x* in the subquery result.

The IN Operator

- <value> IN (<subquery>) is true if and only if the <value> is a member of the relation produced by the subquery.
  - Opposite: <value> NOT IN (<subquery>).
- IN-expressions can appear in WHERE clauses.
- WHERE col IN (value1, value2, …)

IN vs. Join


```
SELECT R.a
FROM R, S
WHERE R.b = S.b;

SELECT R.a
FROM R
WHERE b IN (SELECT b FROM S);
```

IN is a Predicate About R's Tuples


```
SELECT a
FROM R
WHERE b IN (SELECT b FROM S);
```

Two 2's

One loop, over the tuples of R

a	b
1	2
3	4

R

b	c
2	5
2	6

S

(1,2) satisfies the condition; 1 is output once.

This Query Pairs Tuples from R, S


```
SELECT a
FROM R, S
WHERE R.b = S.b;
```

Double loop, over the tuples of R and S

a	b
1	2
3	4

R

b	c
2	5
2	6

S

(1,2) with (2,5) and (1,2) with (2,6) both satisfy the condition; 1 is output twice.

Back to our original query…


```
SELECT bar
FROM Sells
WHERE beer = 'Miller' AND
      price = (SELECT price
               FROM Sells
               WHERE beer = 'Bud');
```

Use IN() or = ANY()

Recap

- IN( ) is equivalent to = ANY( )
- For ANY( ), you can use other comparison operators such as >, <,... etc, but not applicable for IN( )

- The < >ANY operator, however, differs from NOT IN:
  - < >ANY means not = a, or not = b, or not = c
  - NOT IN means not = a, and not = b, and not = c.
  - <>ALL means the same as NOT IN.

Example: =ANY

Sells

Bar	Beer	Price
Jane	Miller	3.00
Joe	Miller	4.00
Joe	Bud	3.00
Jack	Bud	4.00
Tom	Miller	4.50

```
SELECT Bar
FROM Sells
WHERE Beer = 'Miller' AND Price =
        ANY(SELECT Price
            FROM Sells
            WHERE Beer='Bud')
```

Result

Bar
Jane
Joe

The Exists Operator

- EXISTS(<subquery>) is true if and only if the subquery result is not empty.
- Example: From Beers(name, manf) , find those beers that are the unique (only) beer made by their manufacturer.

Example: EXISTS

Notice scope rule: manf refers to closest nested FROM with a relation having that attribute. (Some DBMS consider this ambiguous.)

```
SELECT name
   FROM Beers b1
   WHERE NOT EXISTS (
      SELECT *
      FROM Beers
      WHERE manf = b1.manf AND
            name <> b1.name);
```

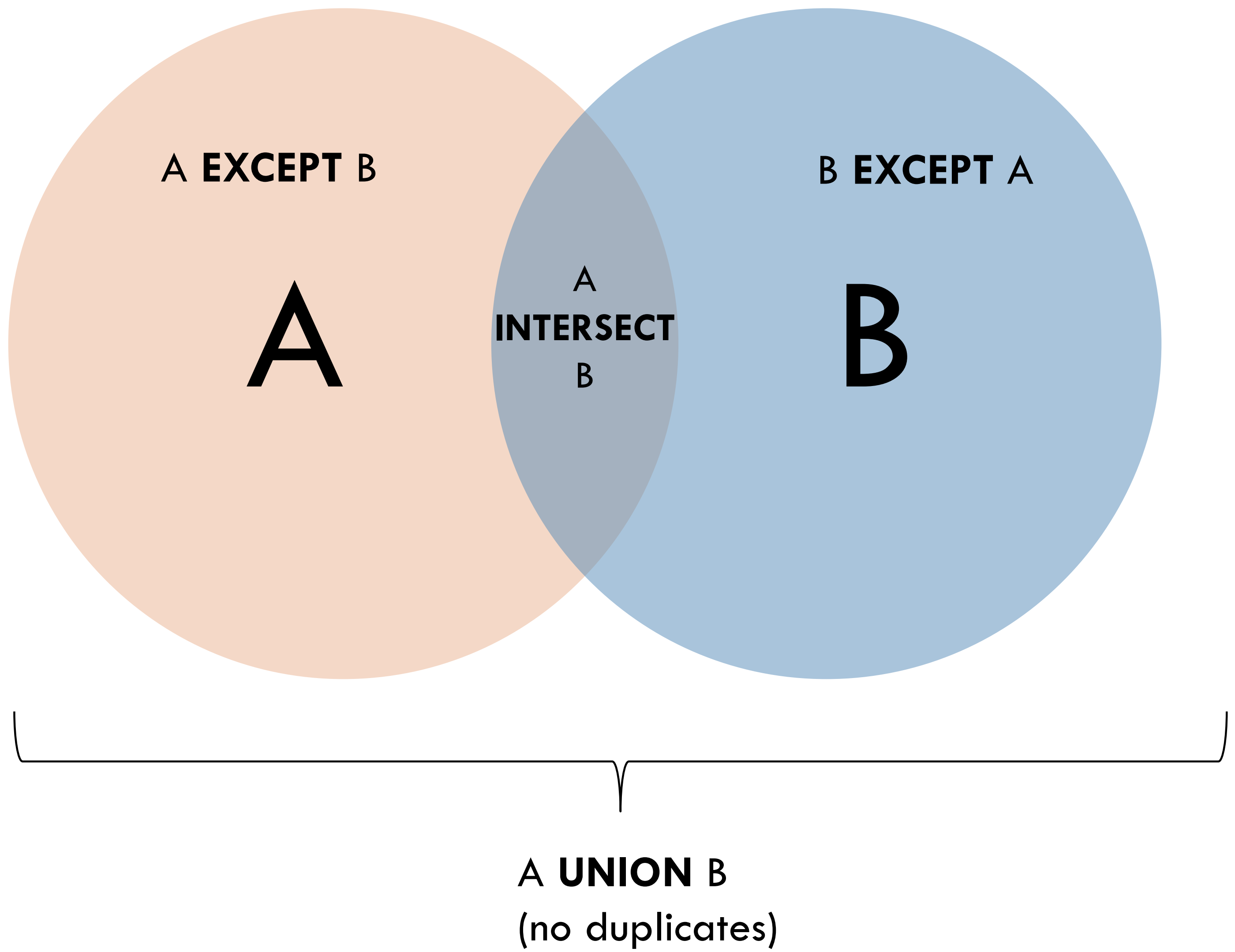
Set of beers with the same manf as b1, but not the same beer

Notice the SQL “not equals” operator

Union, Intersection, and Difference

- Union, intersection, and difference of relations are expressed by the following forms, each involving subqueries:
  - (<subquery>) UNION (<subquery>)
  - (<subquery>) INTERSECT (<subquery>)
  - (<subquery>) EXCEPT (<subquery>)

Visually



Example: Intersection

- Using Likes(drinker, beer), Sells(bar, beer, price), and Frequents(drinker, bar), find the drinkers and beers such that:
  - The drinker likes the beer, and
  - The drinker frequents at least one bar that sells the beer.

Solution

subquery is really a stored table.

```
(SELECT * FROM Likes)
  INTERSECT
(SELECT drinker, beer
 FROM Sells, Frequents
 WHERE Frequents.bar = Sells.bar
);
```

The drinker frequents a bar that sells the beer.

Union, Intersection, and Difference

- Union, intersection, and difference of relations are expressed by the following forms, each involving subqueries:
  - (<subquery>) UNION (<subquery>)
  - (<subquery>) INTERSECT (<subquery>)
  - (<subquery>) EXCEPT (<subquery>)

Example: Intersection

- Using Likes(drinker, beer), Sells(bar, beer, price), and Frequents(drinker, bar), find the drinkers and beers such that:
  - The drinker likes the beer, and
  - The drinker frequents at least one bar that sells the beer.

Solution

subquery is really a stored table.

```
(SELECT * FROM Likes)
  INTERSECT
(SELECT drinker, beer
 FROM Sells, Frequents
 WHERE Frequents.bar = Sells.bar
);
```

The drinker frequents a bar that sells the beer.

Bag Semantics

- A *bag* (or *multiset*) is like a set, but an element may appear more than once.
- Example: {1,2,1,3} is a bag.
- Example: {1,2,3} is also a bag that happens to be a set.

Bag (Multiset) Semantics

- SQL primarily uses bag semantics
- The SELECT-FROM-WHERE statement uses bag semantics
  - originally for efficiency reasons
- The default for union, intersection, and difference is set semantics.
  - That is, duplicates are eliminated as the operation is applied.

Motivation: Efficiency

- When doing projection, it is easier to avoid eliminating duplicates.
  - Just work tuple-at-a-time.
- For intersection or difference, it is most efficient to sort the relations first.
  - At that point you may as well eliminate the duplicates anyway.

Controlling Duplicate Elimination

- Force the result to be a set by SELECT DISTINCT . . .
- Force the result to be a bag (i.e., don't eliminate duplicates) by ALL, as in

  . . . UNION ALL . . .

Example: DISTINCT

- From Sells(bar, beer, price), find all the different prices charged for beers:

```
SELECT DISTINCT price
FROM Sells;
```

Notice that without DISTINCT, each price would be listed as many times as there were bar/beer pairs at that price.

Example: ALL

- Using relations Frequents(drinker, bar) and Likes(drinker, beer):
- Lists drinkers who frequent more bars than they like beers, and do so as many times as the difference of those counts.

```
(SELECT drinker FROM Frequents)
   EXCEPT ALL
(SELECT drinker FROM Likes);
```

Ordering the Display of Tuples

- List in alphabetic order the names of all instructors

  select *name*
  from *instructor*
  order by *name*

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by** *name* **desc**

Humour

SQL query walks into a bar, and approaches two tables and asks, can I join you?

DATABASE MODIFICATIONS

Database Modifications

- A *modification* command does not return a result (as a query does), but changes the database in some way.
- Three kinds of modifications:
  1. *Insert* a tuple or tuples.
  2. *Delete* a tuple or tuples.
  3. *Update* the value(s) of an existing tuple or tuples.

Insertion

- To insert a single tuple:

  INSERT INTO <relation>

  VALUES (<list of values>);

- Example: add to Likes(drinker, beer) the fact that Sally likes Bud.

```
INSERT INTO Likes
VALUES('Sally', 'Bud');
```

Specifying Attributes in INSERT

- We may add to the relation name a list of attributes.
- Two reasons to do so:
  1. We forget the standard order of attributes for the relation.
  2. We don't have values for all attributes, and we want the system to fill in missing components with NULL or a default value.

Example: Specifying Attributes

- Another way to add the fact that Sally likes Bud to Likes(drinker, beer):

```
INSERT INTO Likes(beer, drinker)
VALUES('Bud', 'Sally');
```

Adding Default Values

- In a CREATE TABLE statement, we can follow an attribute by DEFAULT and a value.
- When an inserted tuple has no value for that attribute, the default will be used.

Example: Default Values


```
CREATE TABLE Drinkers (
    name CHAR(30) PRIMARY KEY,
    addr CHAR(50)
        DEFAULT '123 Sesame St.',
    phone CHAR(16)
);
```

Example: Default Values


```
INSERT INTO Drinkers(name)
VALUES('Sally');
```

Resulting tuple:

name	address	phone
Sally	123 Sesame St	NULL

Inserting Many Tuples

- We may insert the entire result of a query into a relation, using the form:

  INSERT INTO <relation>

  (<subquery>);

Example: Insert a Subquery

- Using Frequents(drinker, bar), enter into the new relation Buddies(name) all of Sally's "potential buddies,"
- i.e., those drinkers who frequent at least one bar that Sally also frequents.

```
INSERT INTO Buddies
(SELECT


);
```

Solution

"Those drinkers who frequent at least one bar that Sally also frequents"

The other drinker

```
INSERT INTO Buddies
(SELECT d2.drinker
 FROM Frequents d1, Frequents d2
 WHERE d1.drinker = 'Sally' AND
   d2.drinker <> 'Sally' AND
   d1.bar = d2.bar
);
```

Pairs of Drinker tuples where the first is for Sally, the second is for someone else, and the bars are the same.

Deletion

- To delete tuples satisfying a condition from some relation:

```
DELETE FROM <relation>
WHERE <condition>;
```

Example: Deletion

- Delete from Likes(drinker, beer) the fact that Sally likes Bud:

```
DELETE FROM Likes
WHERE drinker = 'Sally' AND
      beer = 'Bud';
```

Example: Delete all Tuples

- Make the relation Likes empty:

```
DELETE FROM Likes;
```

- Note no WHERE clause needed.

Example: Delete Some Tuples

- Delete from Beers(name, manf) all beers for which there is another beer by the same manufacturer.

```
DELETE FROM Beers b
WHERE
    EXISTS (
    SELECT name
       FROM Beers
    WHERE manf = b.manf AND
         name <> b.name);
```

Beers with the same manufacturer and a different name from the name of the beer represented by tuple b.

Semantics of Deletion --- (1)

- Suppose Anheuser-Busch makes only Bud and Bud Lite.
- Suppose we come to the tuple *b* for Bud first.
- The subquery is nonempty, because of the Bud Lite tuple, so we delete Bud.
- Now, when *b* is the tuple for Bud Lite, do we delete that tuple too?

Semantics of Deletion --- (2)

- Answer: we *do* delete Bud Lite as well.
- The reason is that deletion proceeds in two stages:
  1. Mark all tuples for which the WHERE condition is satisfied.
  2. Delete the marked tuples.

Updates

- To change certain attributes in certain tuples of a relation:

  UPDATE <relation>

  SET <list of attribute assignments>

  WHERE <condition on tuples>;

Example: Update

- Change drinker Fred’s phone number to 555-1212:

```
UPDATE Drinkers
SET phone = ’555-1212’
WHERE name = ’Fred’;
```

Example: Update Several Tuples

- Make $4 the maximum price for beer:

```
UPDATE Sells
SET price = 4.00
WHERE price > 4.00;
```

AGGREGATION, GROUPING & OUTER JOINS

Aggregation

- SUM, AVG, COUNT, MIN, and MAX can be applied to a column in a SELECT clause to produce that aggregation on the column.
- COUNT(*) counts the number of tuples.

Example: Aggregation

- From Sells(bar, beer, price), find the average price of Bud:

```
SELECT AVG(price)
FROM Sells
WHERE beer = 'Bud';
```

Eliminating Duplicates in an Aggregation

- Use DISTINCT inside an aggregation.
- Example: find the number of *different* prices charged for Bud:

```
SELECT COUNT(DISTINCT price)
  FROM Sells
  WHERE beer = 'Bud';
```

NULL's Ignored in Aggregation

- NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column.
- But if all the values in a column are NULL, then the result of the aggregation is NULL.
  - Exception: COUNT of an empty set is 0.

Example: Effect of NULL's

Sells(bar, beer, price)

```
SELECT count(*)
FROM Sells
WHERE beer = 'Bud';
```

The number of bars that sell Bud.

```
SELECT count(price)
FROM Sells
WHERE beer = 'Bud';
```

The number of bars that sell Bud at a known price (i.e., where price is not NULL)

Example Query

- Find the age of the youngest employee at each rating level

SELECT MIN (age)

FROM Employees

WHERE rating = i

Grouping

- We may follow a SELECT-FROM-WHERE expression by GROUP BY and a list of attributes.
- The relation that results from the SELECT-FROM-WHERE is grouped according to the values of all those attributes, and any aggregation is applied only within each group.

```
SELECT    rating, MIN(age)
FROM      Employees
GROUP BY  rating
```

Example: Grouping

- From Sells(bar, beer, price), find the average price for each beer:

```
SELECT beer, AVG(price)
FROM Sells
GROUP BY beer;
```

beer	AVG(price)
Bud	2.33
Miller	4.55
…	…

Example: Grouping

- From Sells(bar, beer, price) and Frequents(drinker, bar), find for each drinker the average price of Bud at the bars they frequent:

```
SELECT drinker, AVG(price)
FROM Frequents, Sells
WHERE beer = 'Bud' AND
      Frequents.bar = Sells.bar
GROUP BY drinker;
```

Compute all drinker-bar-price triples for Bud.

Then group them by drinker.

Restriction on SELECT Lists With Aggregation

- If any aggregation is used, then each element of the SELECT list must be either:
  1. Aggregated, or
  2. An attribute on the GROUP BY list.

Illegal Query Example


```
SELECT bar, beer, AVG(price)
FROM Sells
GROUP BY bar
```

- But this query is illegal in SQL.
- Only one tuple output for each bar, no unique way to select which beer to output

A Closer Look

SELECT bar, beer, AVG(price) AS avgP
FROM Sells
GROUP BY bar beer

Sells

Bar	Beer	Price
Joe	Bud	3.00
Joe	Miller	4.00
Tom	Bud	3.50
Tom	Miller	4.25
Jane	Bud	3.25
Jane	Miller	4.75
Jane	Coors	4.00

Result

bar	beer	avgP
Joe	?	3.50
Tom	?	3.88
Jane	?	4.00

{Bud, Miller, Coors}?

Only one tuple output for each bar, no unique way to select which beer to output

HAVING Clauses

- HAVING <condition> may follow a GROUP BY clause.
- If so, the condition applies to each group, and groups not satisfying the condition are eliminated.

Example: HAVING

- From Sells(bar, beer, price) and Beers(name, manf), find the average price of those beers that are either served in at least three bars or are manufactured by Pete's.

Solution

Sells(bar, beer, price) and Beers(name, manf),


```
SELECT beer, AVG(price)
FROM Sells
GROUP BY beer
HAVING COUNT(bar) >= 3 OR
      beer IN (SELECT name
                FROM Beers
                WHERE manf = 'Pete''s');
```

Beer groups with at least 3 non-NULL bars

Beers manu-factured by Pete's.

Requirements on HAVING Conditions

- Anything goes in a subquery.
- Outside subqueries, they may refer to attributes only if they are either:
  1. A grouping attribute, or
  2. Aggregated

  (same condition as for SELECT clauses with aggregation).

A Final Example


```
SELECT Bar, SUM(Qty) AS sumQ
FROM Sells
GROUP BY Bar
HAVING sum(Qty) > 4
```

Sells

Bar	Beer	Price	Qty
Joe	Bud	3.00	2
Joe	Miller	4.00	2
Tom	Bud	3.50	1
Tom	Miller	4.25	4
Jane	Bud	3.25	1
Jane	Miller	4.75	3
Jane	Coors	4.00	2

Result

Bar	sumQ
Tom	5
Jane	6

Assignment 2

- Due Nov 4th at 10:00pm
  - Populate database early!
  - Practice SQL queries as prep for midterm

- This week: outerjoins, views, indexes
- Midterm material cut-off this Thursday's lecture
- Review midterm practice questions: Oct 21, 23

Example: Grouping

- From Sells(bar, beer, price), find the average price for each beer:

```
SELECT beer, AVG(price)
FROM Sells
GROUP BY beer;
```

beer	AVG(price)
Bud	2.33
Miller	4.55
…	…

Restriction on SELECT Lists With Aggregation

- If any aggregation is used, then each element of the SELECT list must be either:
  1. Aggregated, or
  2. An attribute on the GROUP BY list.

Illegal Query Example


```
SELECT bar, beer, AVG(price)
FROM Sells
GROUP BY bar
```

- But this query is illegal in SQL.
- Only one tuple output for each bar, no unique way to select which beer to output

A Closer Look

SELECT bar, beer, AVG(price) AS avgP
FROM Sells
GROUP BY bar beer

Sells

Bar	Beer	Price
Joe	Bud	3.00
Joe	Miller	4.00
Tom	Bud	3.50
Tom	Miller	4.25
Jane	Bud	3.25
Jane	Miller	4.75
Jane	Coors	4.00

Result

bar	beer	avgP
Joe	?	3.50
Tom	?	3.88
Jane	?	4.00

{Bud, Miller, Coors}?

Only one tuple output for each bar, no unique way to select which beer to output

Example: HAVING

- From Sells(bar, beer, price) and Beers(name, manf), find the average price of those beers that are either served in at least three bars or are manufactured by Pete's.

Solution

Sells(bar, beer, price) and Beers(name, manf),


```
SELECT beer, AVG(price)
FROM Sells
GROUP BY beer
HAVING COUNT(bar) >= 3 OR
    beer IN (SELECT name
        FROM Beers
        WHERE manf = 'Pete''s');
```

Beer groups with at least 3 non-NULL bars

Beers manu-factured by Pete's.

Requirements on HAVING Conditions

- Anything goes in a subquery.
- Outside subqueries, they may refer to attributes only if they are either:
  1. A grouping attribute, or
  2. Aggregated

  (same condition as for SELECT clauses with aggregation).

Cross Product

- Evaluating joins involves combining two or more relations
- Given two relations, S and R, each row of S is paired with each row of R
- Result schema: one attribute from each attribute of S and R

Example

Sells

Bar	Beer	Price
Joe	Bud	3.00
Tom	Miller	4.00
Jane	Lite	3.25

Frequents

Drinker	Bar
Aaron	Joe
Mary	Jane

Cross product, also known as the Cartesian product

Sells x Frequents

(Bar)	Beer	Price	Drinker	(Bar)
Joe	Bud	3.00	Aaron	Joe
Joe	Bud	3.00	Mary	Jane
Tom	Miller	4.00	Aaron	Joe
Tom	Miller	4.00	Mary	Jane
Jane	Lite	3.25	Aaron	Joe
Jane	Lite	3.25	Mary	Jane

```
SELECT drinker
FROM Frequents, Sells
WHERE beer = 'Bud' AND
      Frequents.bar = Sells.bar
```

Drinker
Aaron

Joined Relations

- **Join operations** take two relations and return as a result another relation.

- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join

Join Operations – Example

- Relation *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that

  prereq information is missing for CS-315 and

  course information is missing for CS-347

Outer Join

- An extension of the join operation that avoids loss of information.
- Suppose you have two relations R and S. A tuple of *R* that has no tuple of *S* with which it joins is said to be *dangling*.
  - Similarly for a tuple of *S*.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Outerjoin preserves dangling tuples by padding them with NULL.

Left Outer Join

- *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- *prereq*

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- *course* **left outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

Right Outer Join

- *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- *prereq*

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- *course* **right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

Full Outer Join

- *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- *prereq*

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- *course* **full outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Inner Join

- *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- *prereq*

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- *course* **inner join** *prereq* **on** *course.course_id* = *prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101

Outerjoins

- R OUTER JOIN S is the core of an outerjoin expression. It is modified by:
  1. Optional NATURAL in front of OUTER.
     - Check equality on all common attributes
     - No two attributes with the same name in the output
  2. Optional ON <condition> after JOIN.
  3. Optional LEFT, RIGHT, or FULL before OUTER.
     - LEFT = pad dangling tuples of R only.
     - RIGHT = pad dangling tuples of S only.
     - FULL = pad both; this choice is the default.

Credit: Renee J. Miller

Example: Outerjoin

R =

A	B
1	2
4	5

S =

B	C
2	3
6	7

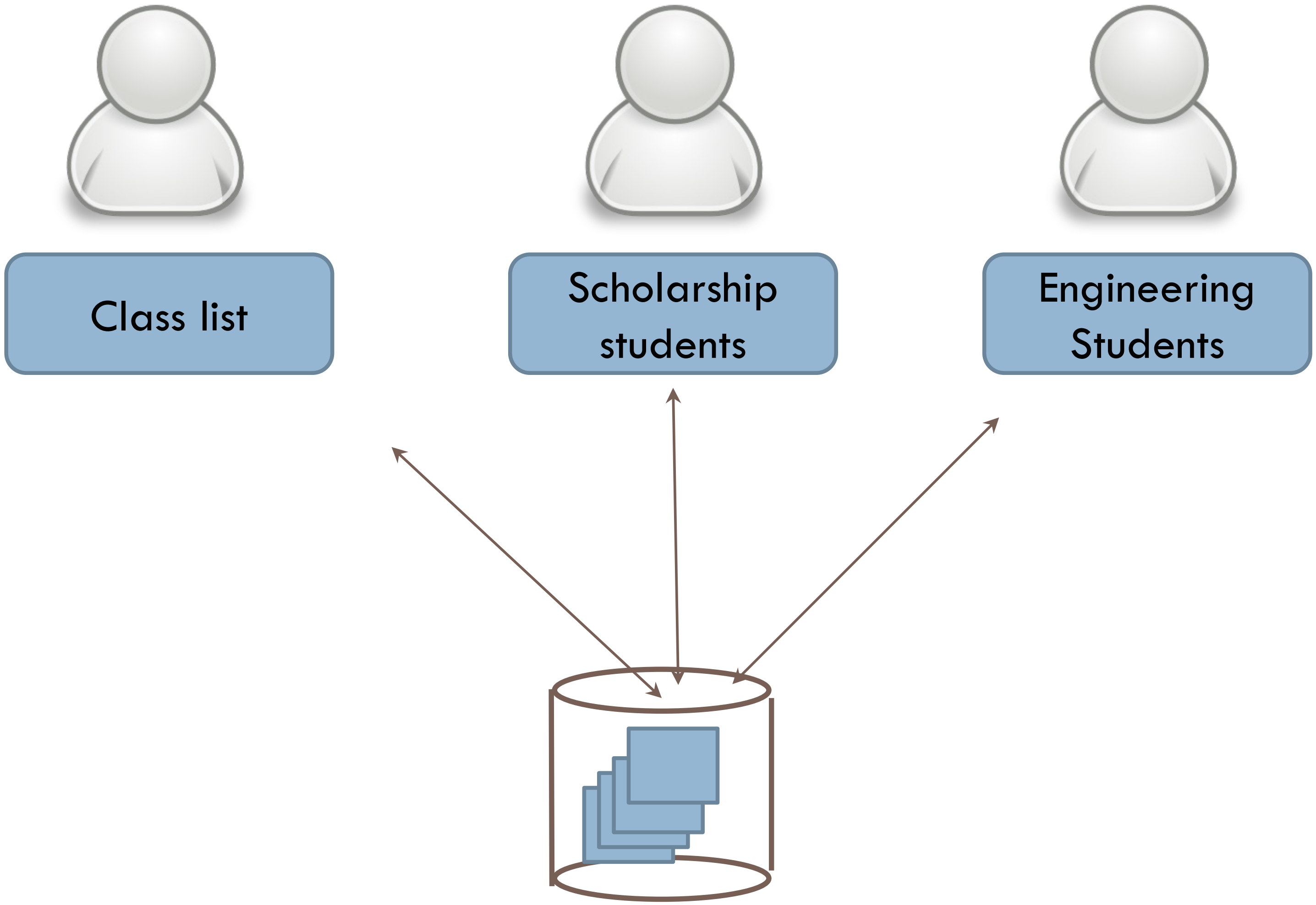
(1,2) joins with (2,3), but the other two tuples are dangling.

R NATURAL FULL OUTERJOIN S =

A	B	C
1	2	3
4	5	NULL
NULL	6	7

VIEWS

Scenario

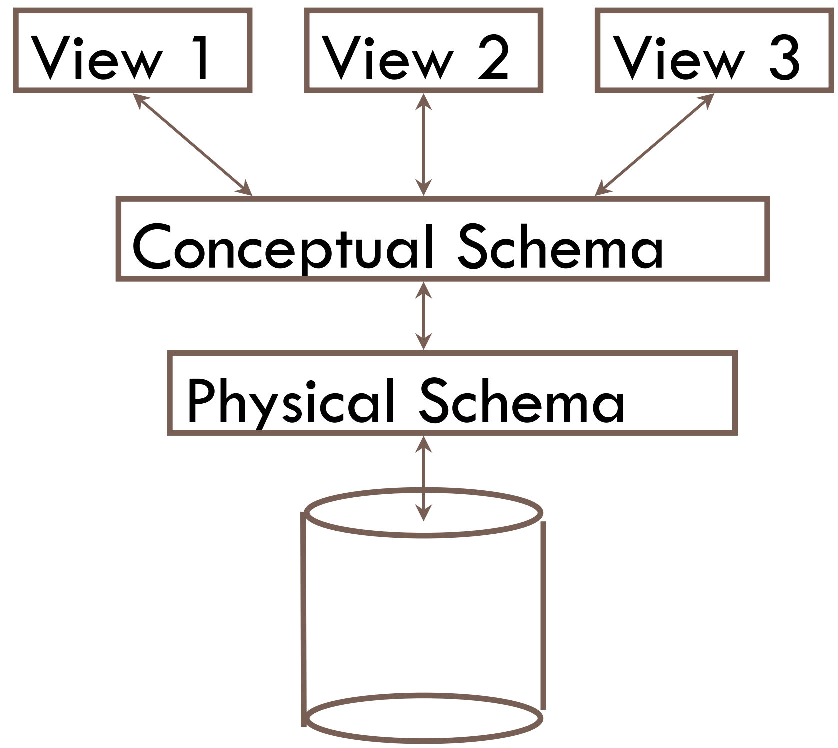


Views

- In most cases, it is not desirable for all users to see the entire data instance.
- A **view** provides a mechanism to hide certain data from the view of certain users.

Levels of Abstraction

- Many *views*, single *conceptual (logical) schema* and *physical schema*.
  - Views describe how users see the data.
  - Conceptual schema defines logical structure
  - Physical schema describes the files and indexes used.



Credit: Renee J. Miller

Views

- A *view* is a relation defined in terms of stored tables (called *base tables*) and other views.
- Two kinds:
  1. *Virtual* = not stored in the database; just a query for constructing the relation.
  2. *Materialized* = actually constructed and stored.

Declaring Views

- Declare by:

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

- A view name
- A possible list of attribute names (for example, 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

- Default is virtual.

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.
  - Also: a limited ability to modify views if it makes sense as a modification of one underlying base table.
- Example query:

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

Another Example

- Example: View Synergy has (drinker, beer, bar) triples such that the bar serves the beer, the drinker frequents the bar and likes the beer.

Example: The View

Pick one copy of each attribute

```
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;
```

Natural join of Likes, Sells, and Frequents

Updates on Views

- Generally, it is impossible to modify a virtual view, because it doesn't exist.
- Can't we "translate" updates on views into "equivalent" updates on base tables?
  - Not always (in fact, not often)
  - Most systems prohibit most view updates
- We cannot insert into Synergy --- it is a virtual view.

Interpreting a View Insertion

- But we could try to translate a (drinker, beer, bar) triple into three insertions of projected pairs, one for each of Likes, Sells, and Frequents.

Insertion


```
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);
```

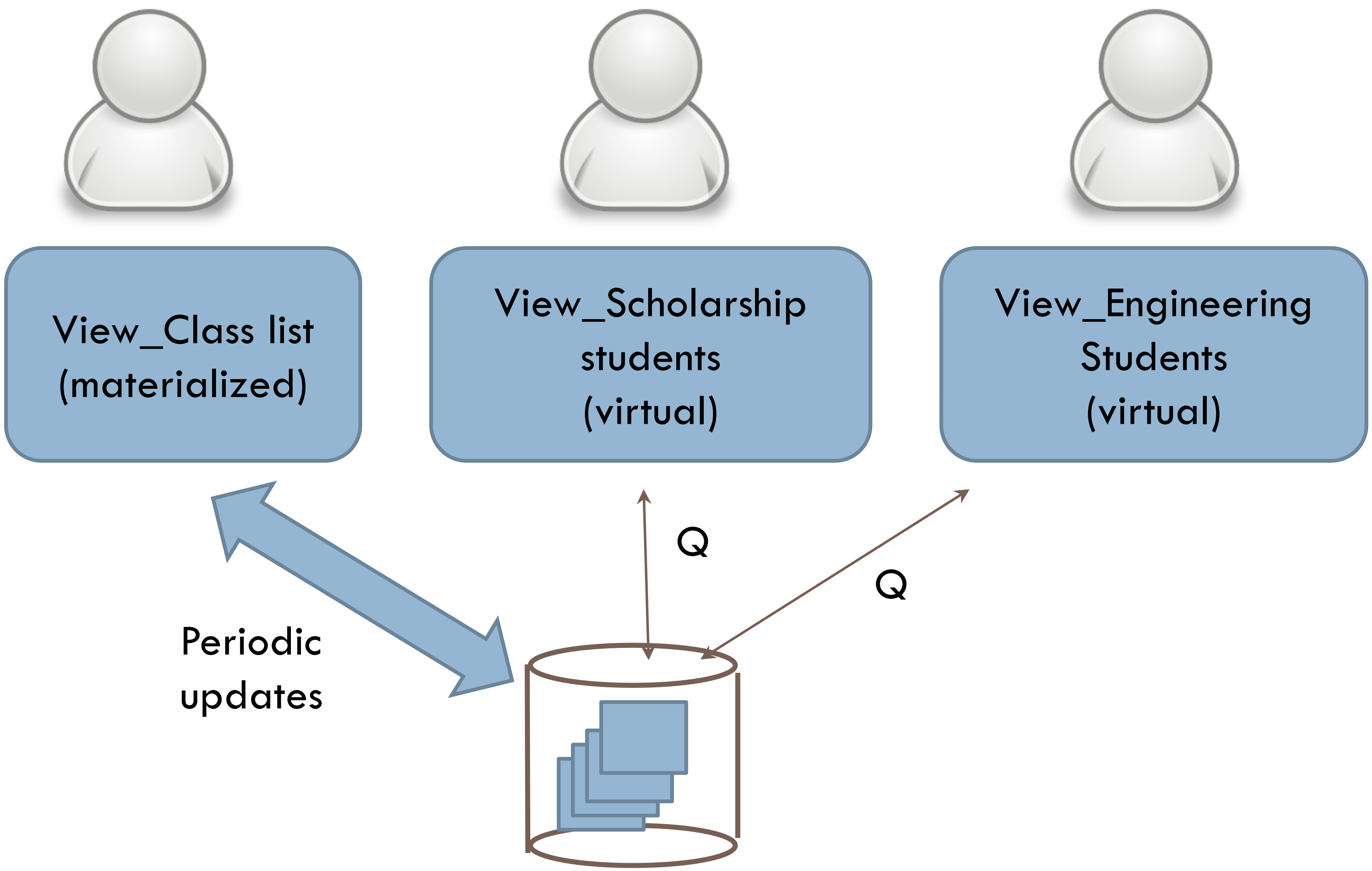
- Sells.price will have to be NULL.
- There isn't always a unique translation.

Materialized Views

- *Materialized* = actually constructed and stored (keeping a temporary table)

- Concerns: maintaining correspondence between the base table and the view when the base table is updated
- Strategy: incremental update

Example


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

Views

- A *view* is a relation defined in terms of stored tables (called *base tables*) and other views.
- Two kinds:
  1. *Virtual* = not stored in the database; just a query for constructing the relation.
  2. *Materialized* = actually constructed and stored.

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.
  - Also: a limited ability to modify views if it makes sense as a modification of one underlying base table.
- Example query:

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

Another Example

- Example: View Synergy has (drinker, beer, bar) triples such that the bar serves the beer, the drinker frequents the bar and likes the beer.

Example: The View

Pick one copy of each attribute

```
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;
```

Natural join of Likes, Sells, and Frequents

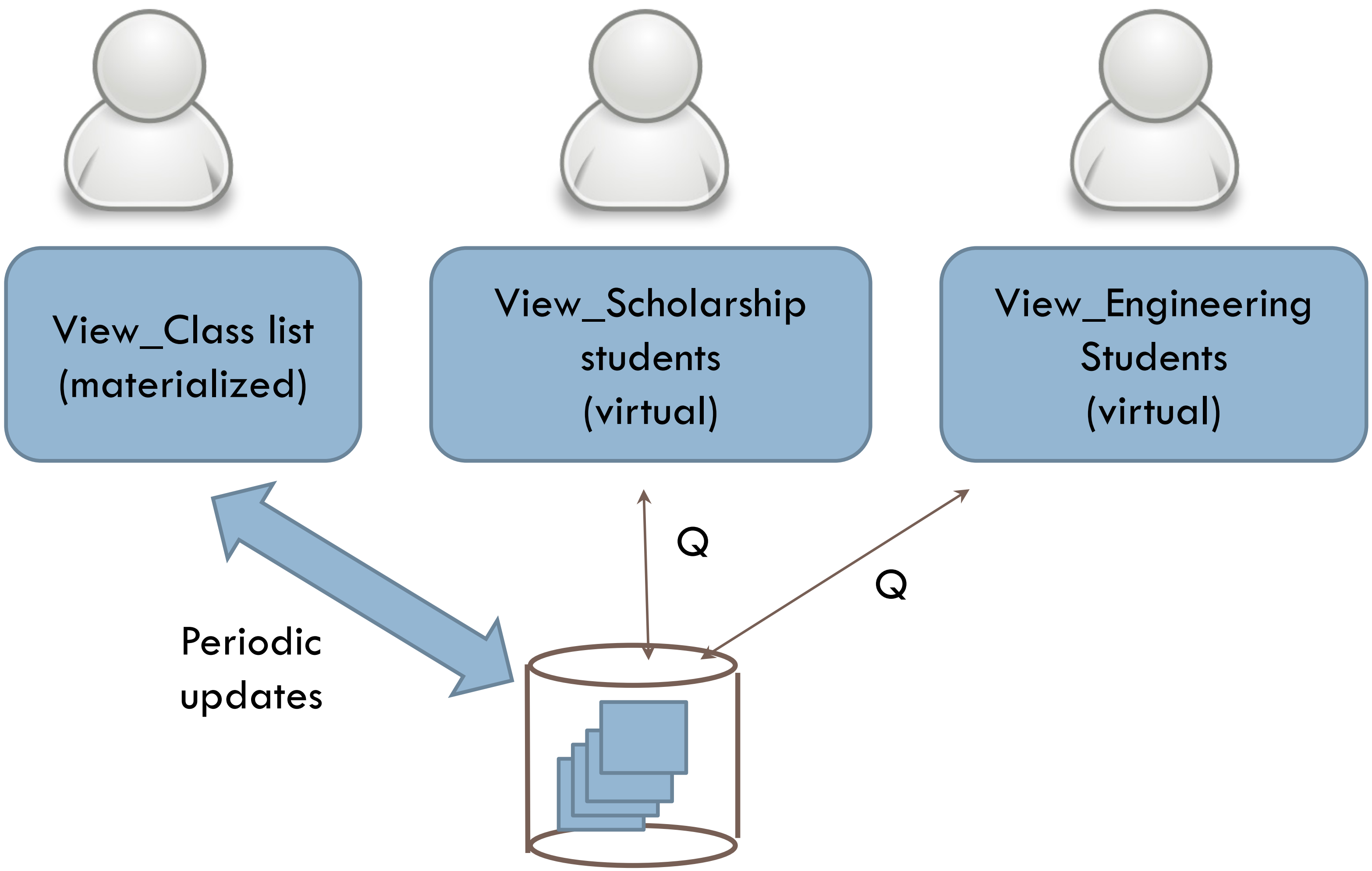
Updates on Views

- Generally, it is impossible to modify a virtual view, because it doesn't exist.
- Can't we "translate" updates on views into "equivalent" updates on base tables?
  - Not always (in fact, not often)
  - Most systems prohibit most view updates
- We cannot insert into Synergy --- it is a virtual view.

Materialized Views

- *Materialized* = actually constructed and stored (keeping a temporary table)
- Concerns: maintaining correspondence between the base table and the view when the base table is updated
- Strategy: incremental update

Example


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

Materialized View Updates

- Update on a single view without aggregate operations: update may map to an update on the underlying base table (most SQL implementations)

- Views involving joins: an update *may map to an* update on the underlying base relations not always possible

INDEXES

Example

- Find the price of beers manufactured by Pete’s and sold by Joe.

SELECT price

FROM Beers, Sells

WHERE manf = ‘Pete’’s’ AND bar = ‘Joe’ AND

Sells.beer = Beers.name

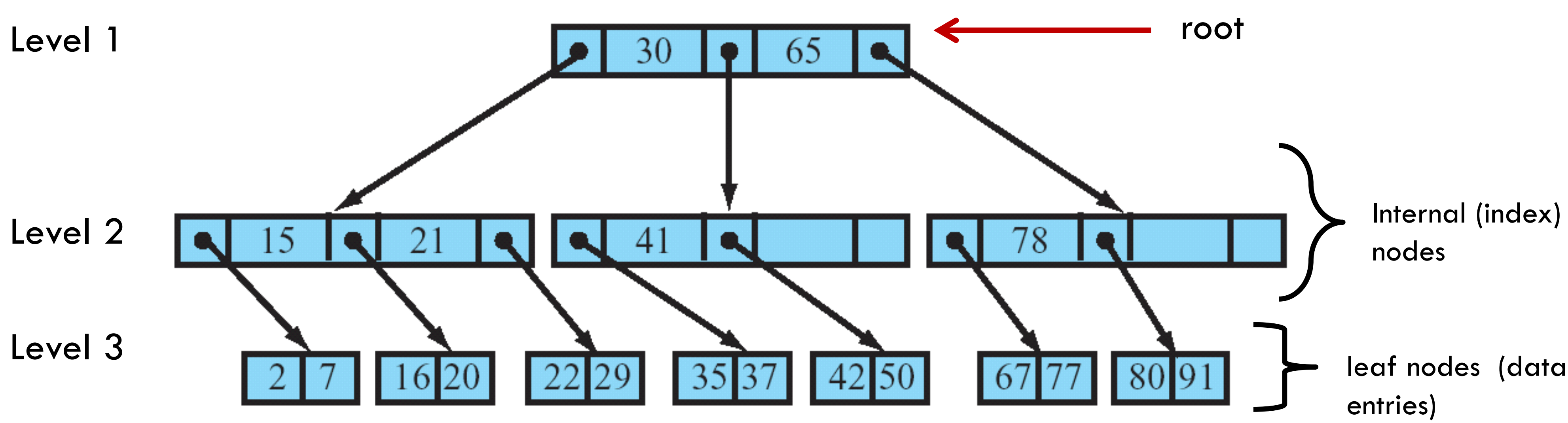
An Index

- A data structure used to speed access to tuples of a relation, based on values of one or more attributes ("search key" fields)
- Organizes records via trees or hashing
- Given a value *v*, the index takes us to only those tuples that have *v* in the attribute(s) of the index.

- Example: use BeerInd (on manf) and SellInd (on bar, beer) to find the prices of beers manufactured by Pete's and sold by Joe.

B+ Tree Index

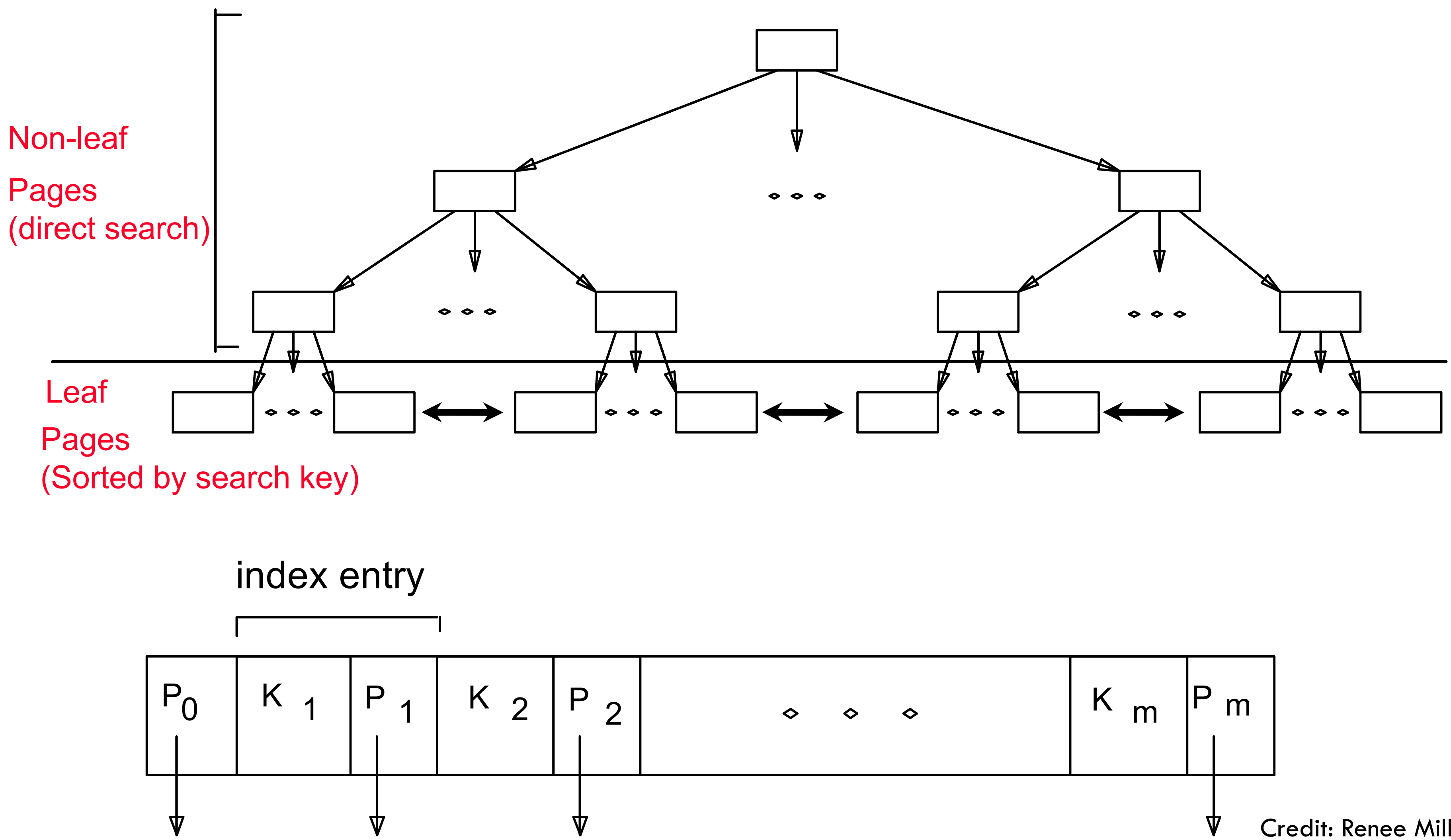
- The B+ tree structure is the most common index type in databases
- Index files can be quite large, often stored on disk, partially loaded into memory as needed
- Each node is at least 50% full



Credit: S. Lee

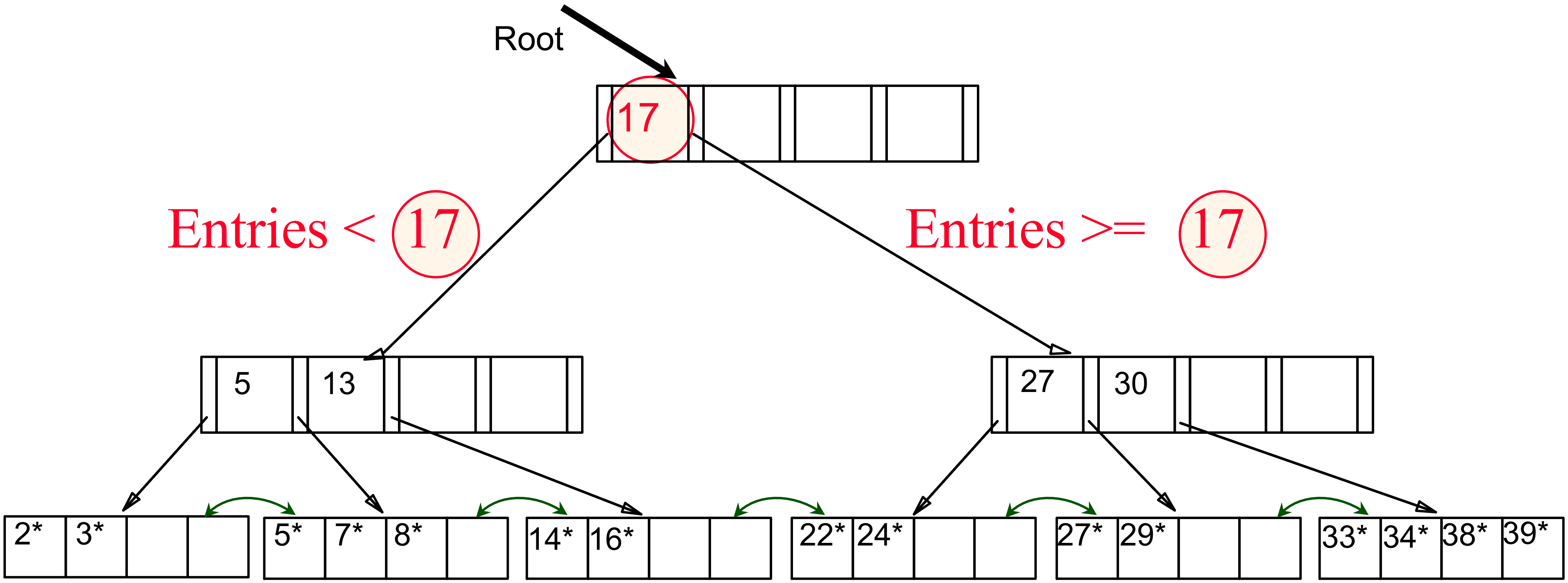
B+ Tree Index

Supports equality and range-searches efficiently



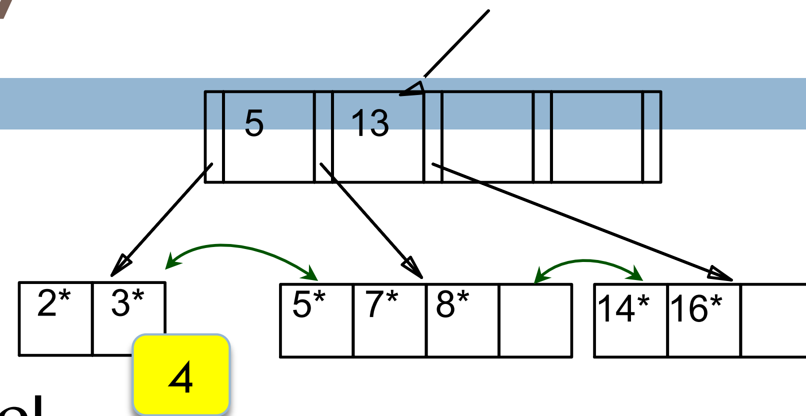
Credit: Renee Miller

Example



- Find 28*? 29*? All > 15* and < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree

Inserting a Data Entry

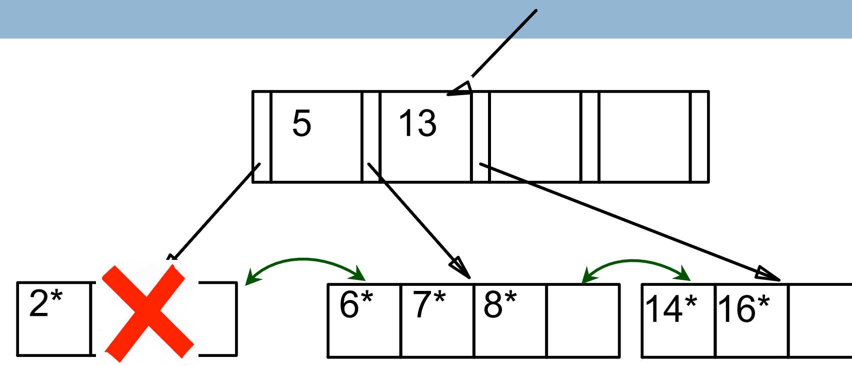


- Find correct leaf L.
- Put data entry onto L.
  - If L has enough space, done!
  - Else, must *split* L (into L and a new node L2)
    - Redistribute entries evenly, copy up middle key.
    - Insert index entry pointing to L2 into parent of L.
- This can happen recursively
  - To split index node, redistribute entries evenly, but push up middle key.
- Splits "grow" tree; root split increases height.

Insert data value 4

Deleting a Data Entry

Delete value 3

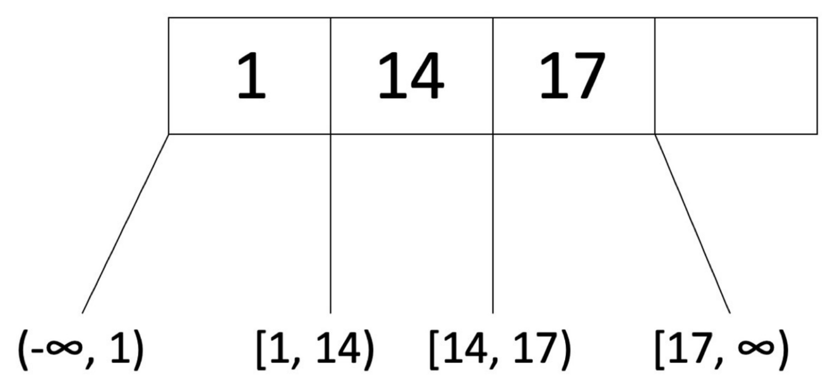


- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, done!
  - If not,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.

B+ Tree: Most Widely Used Index

- Insert/delete at $\log_F N$ cost; keep tree *height-balanced.* (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains **d** <= *m* <= 2**d** entries. The parameter **d** is called the *order* of the tree.
- Node with order d = 2,

e.g., 2 <= m <= 4



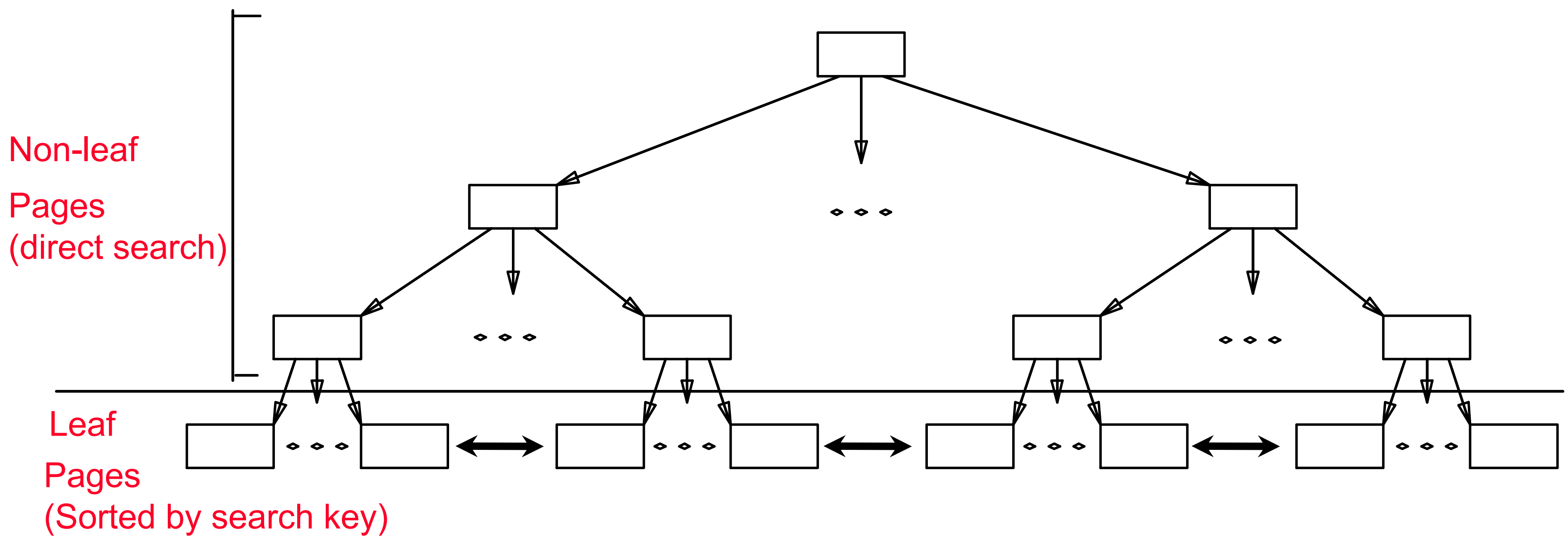
B+ Trees in Practice

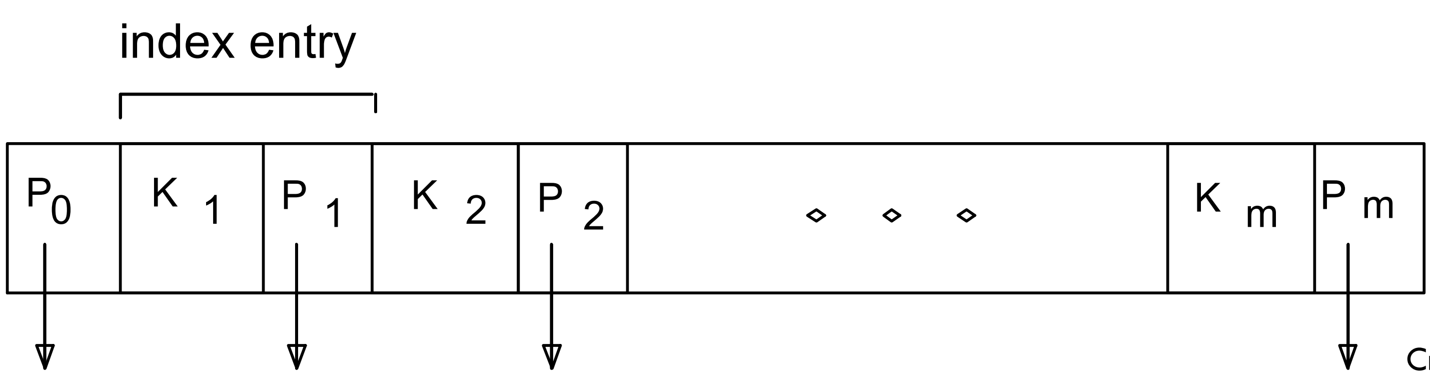
- Typical order: 100.
- Typical fill-factor: ln 2 = 66.5% (approx)
  - average fanout = 2 x 100 x 66.5% = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,721 pages
  - Height 3: 133^3 = 2,352,637 pages

- For typical orders (d ~ 100-200), a shallow B+ tree can accommodate very large files.

B+ Tree Index

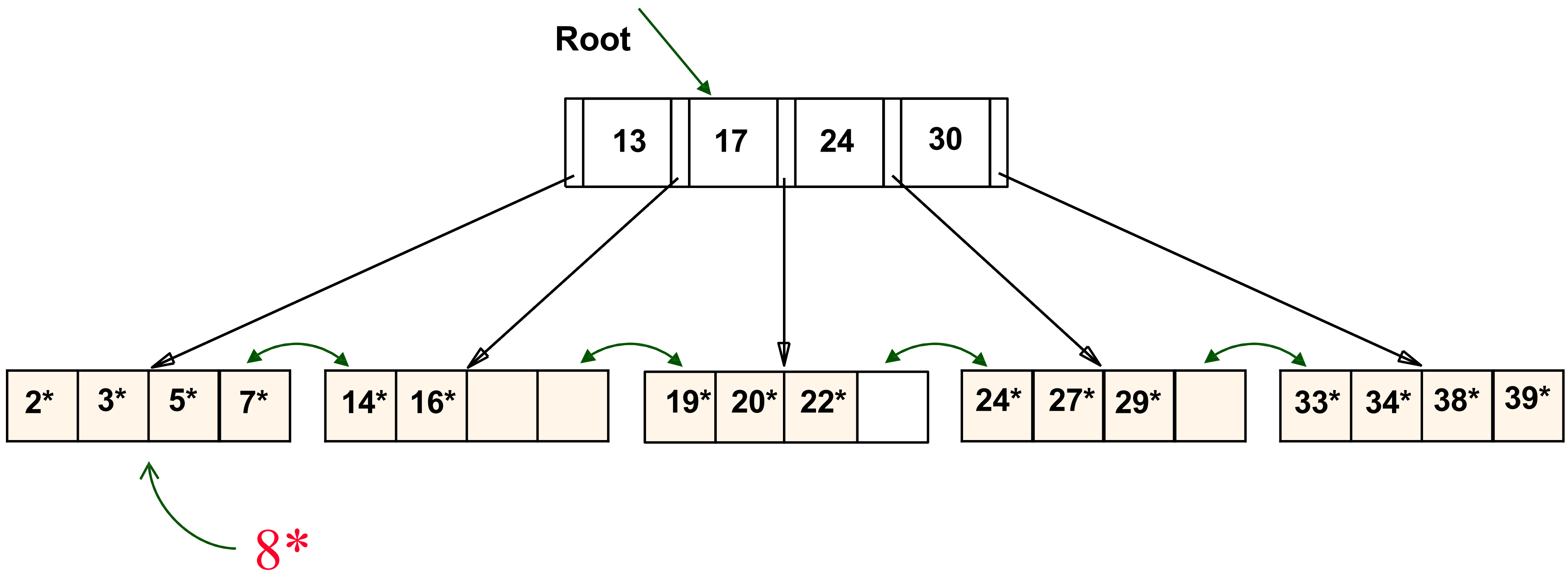
Supports equality and range-searches efficiently



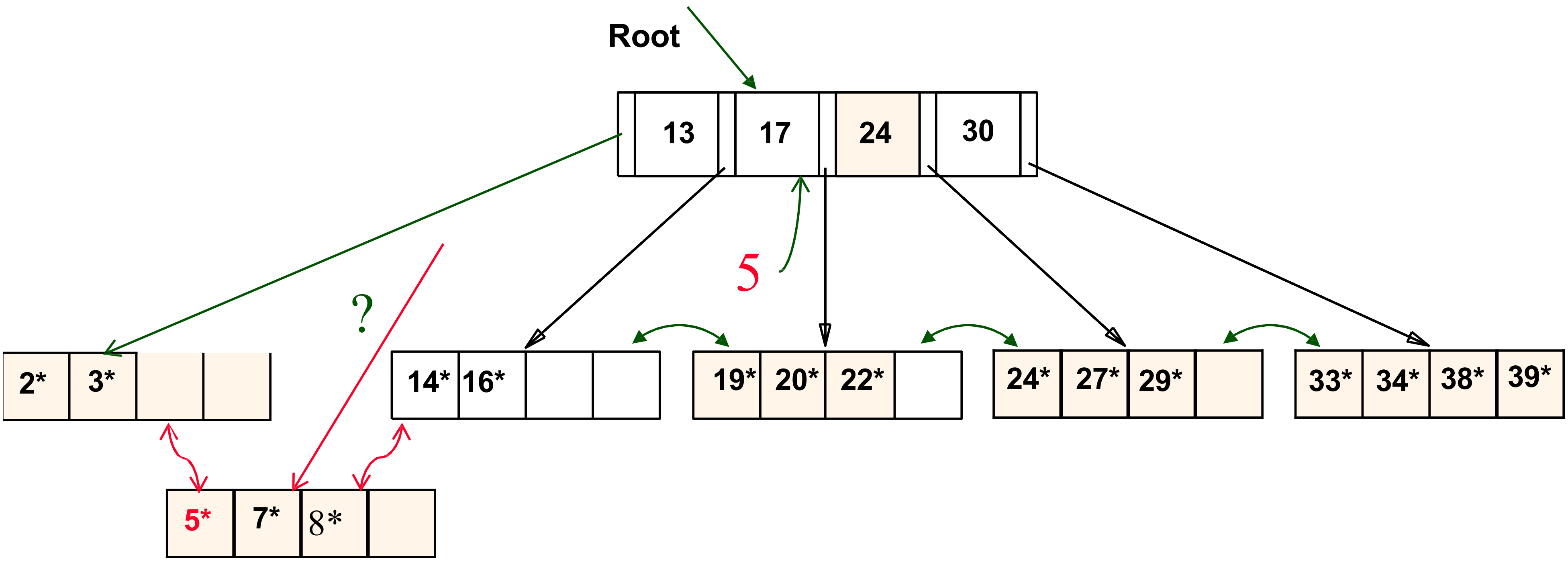


Credit: Renee Miller

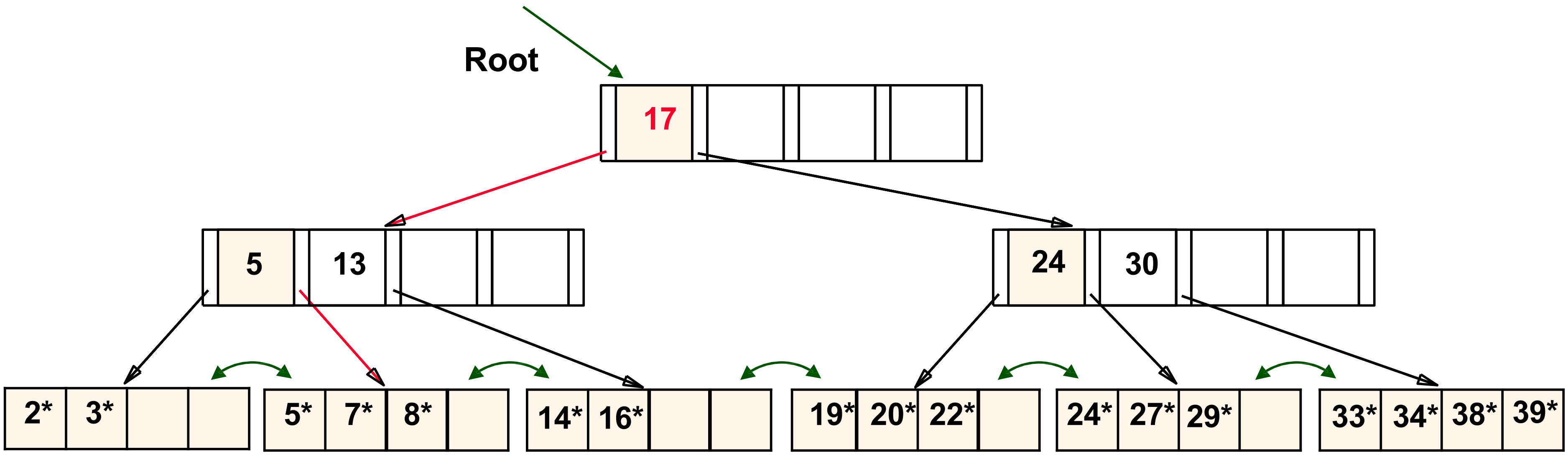
Insertion Example



Insertion Example



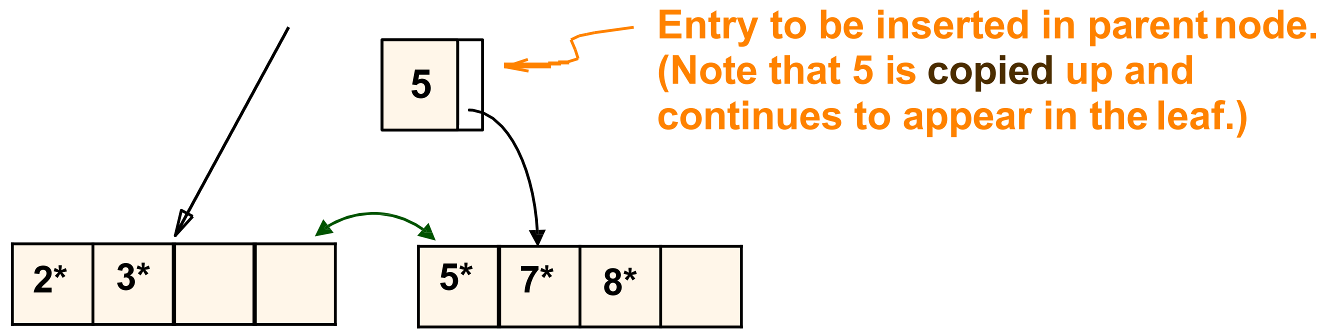
After Inserting 8*

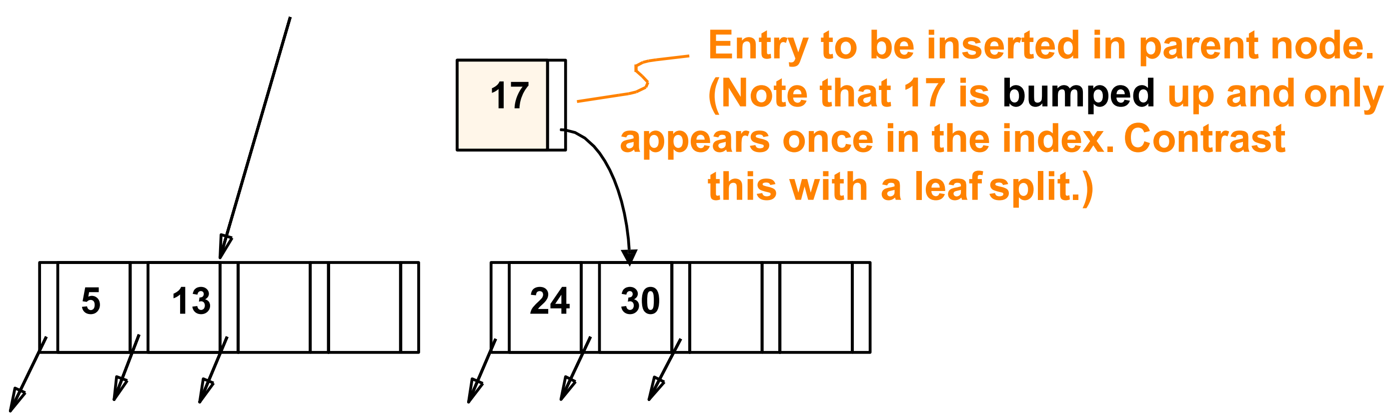


❖ Notice that root was split, leading to increase in height.

Copy-Up vs. Bump-Up

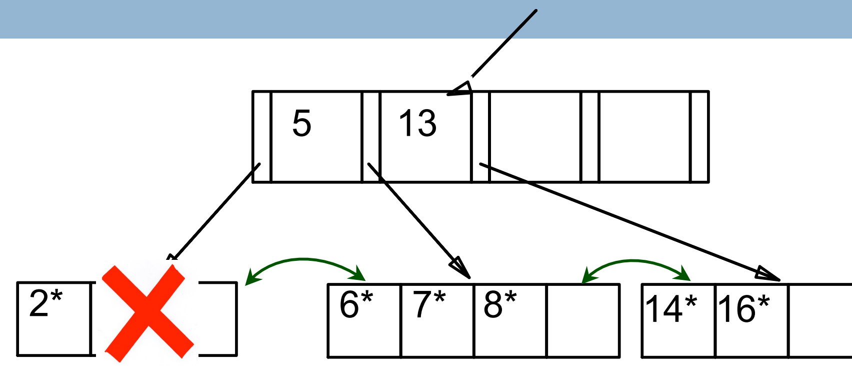
- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between *copy-up* and *bump-up;* Why do we handle leaf page split and index page split differently?





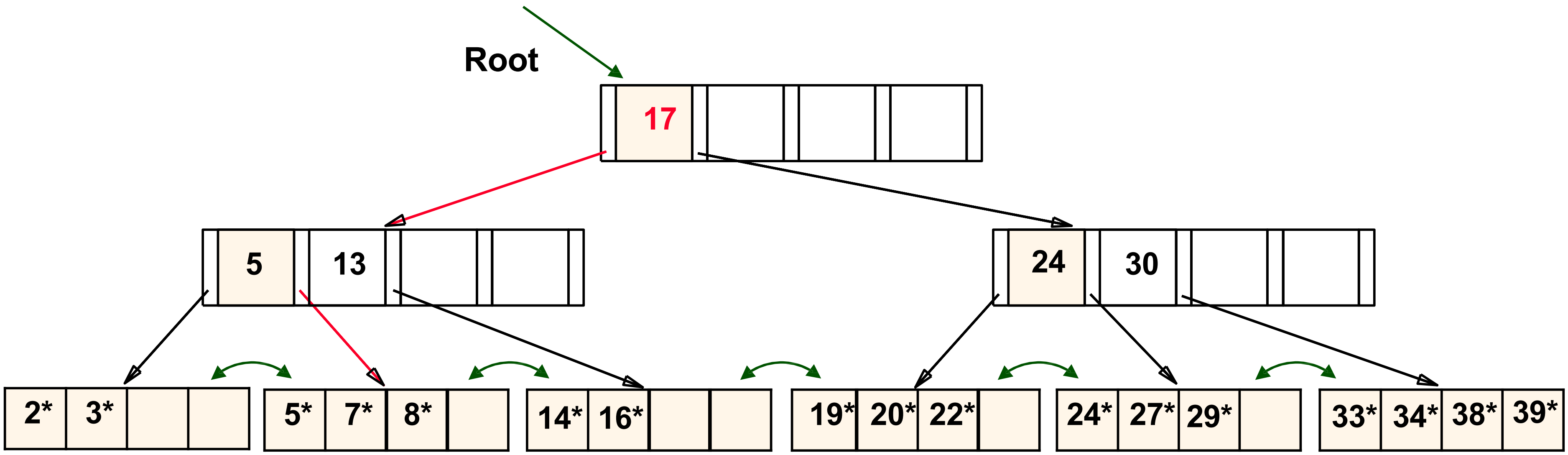
Deleting a Data Entry

Delete value 3



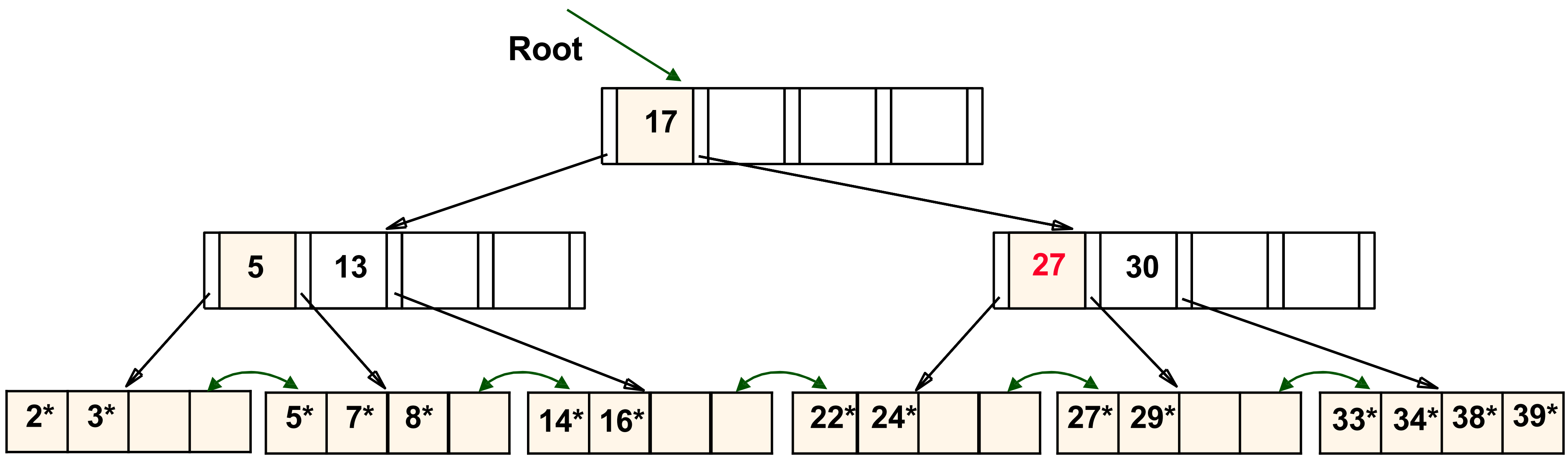
- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, done!
  - If not,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.

Deleting 19* is Straightforward



- What happens if we delete 20* next?

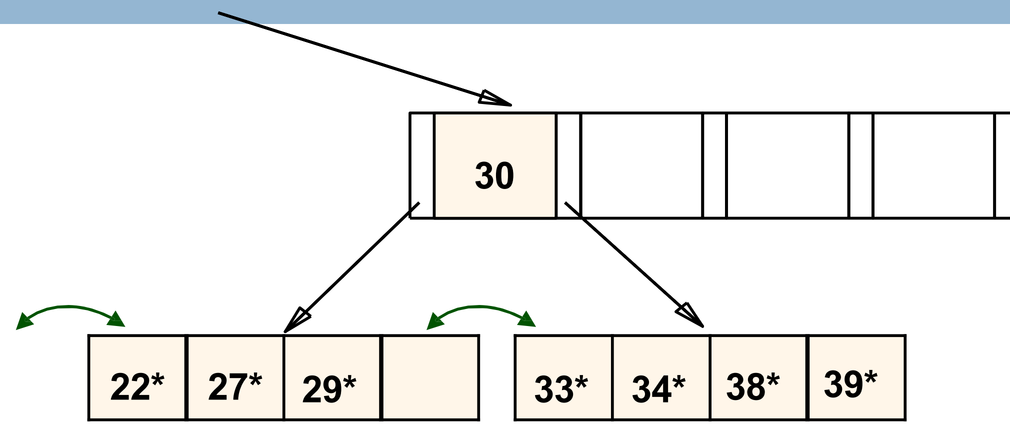
Example Tree: Deleting 19* and 20*

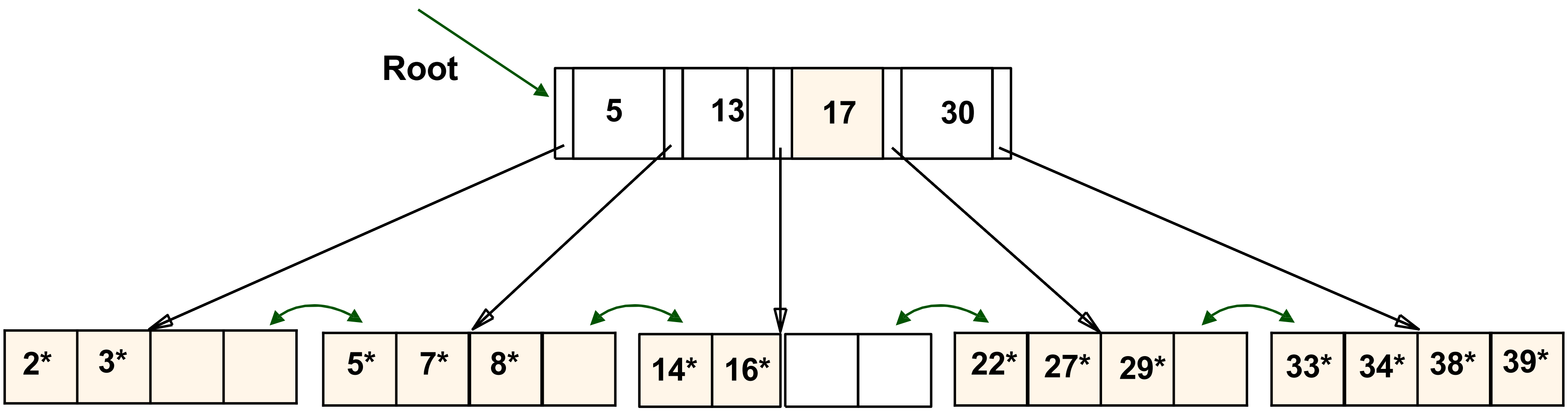


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how *new* middle key is *copied up*.
- What happens if we delete 24* now?

Deleting 24* ...

- Must merge.
- Observe *`toss'* of index entry (on right), and *`pull down'* of index entry (below).

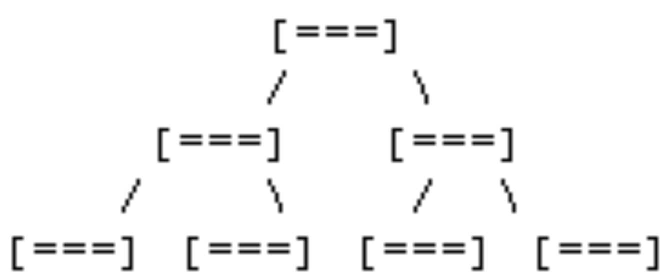




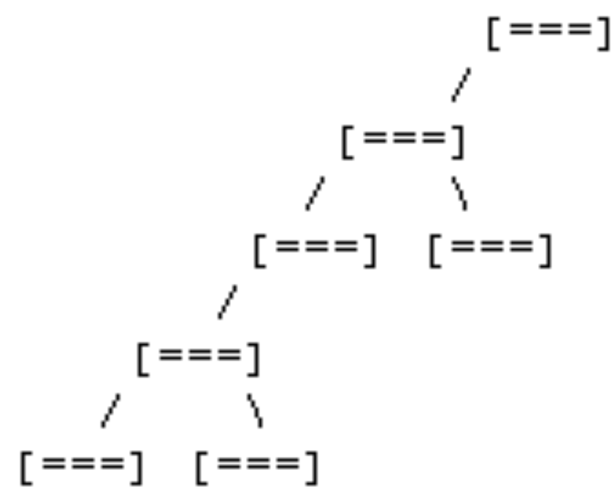
Balanced vs. Unbalanced Trees

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

- Balanced



- Unbalanced

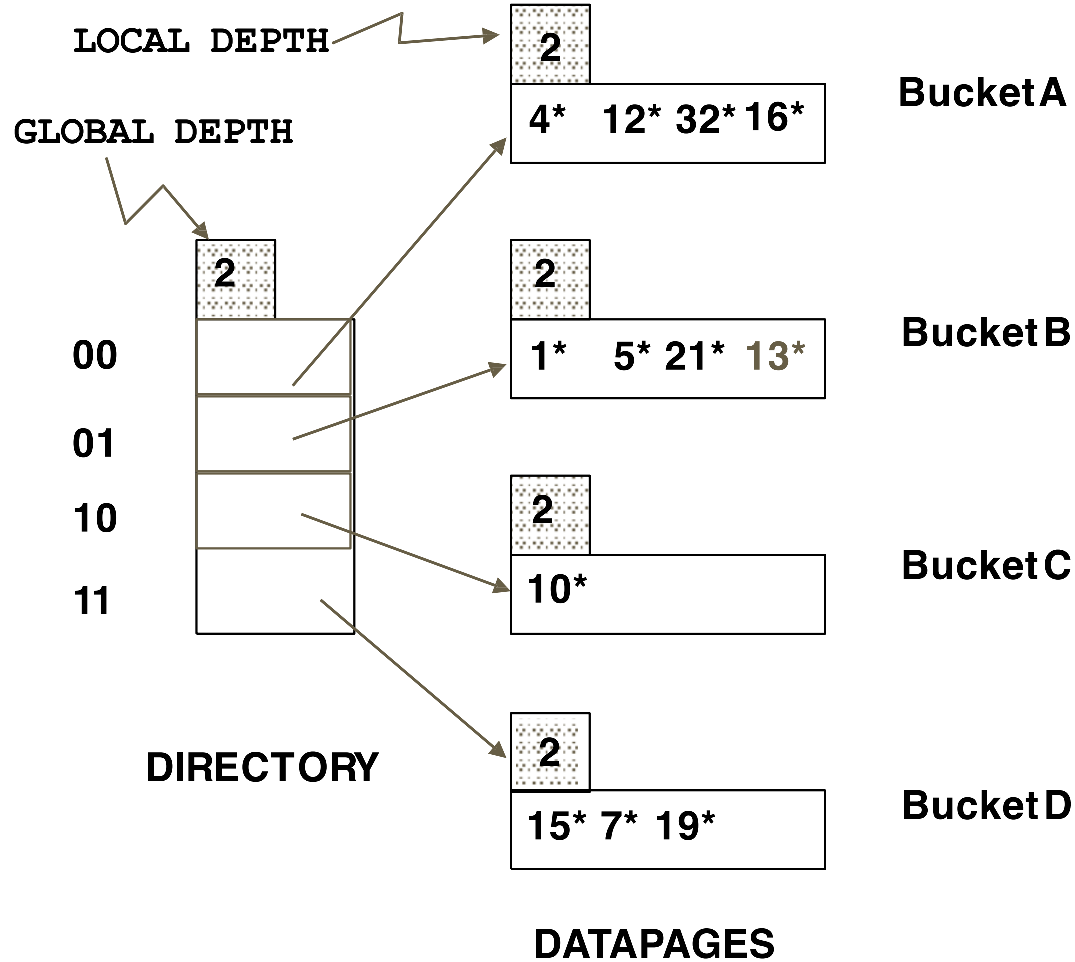


Credit: S. Lee

Hash Based Indexes

- Good for equality searches
- Your index is a collection of *buckets* (bucket = page)
- Define a hash function, *h*, that maps a key to a bucket.
- Store the corresponding data in that bucket.
- Collisions
  - Multiple keys hash to the same bucket.
  - Store multiple keys in the same bucket.
- What do you do when buckets fill?
  - Chaining: link new pages(overflow pages) off the bucket.

Example



- Directory is array of size 4.
- To find bucket for *r*, take last \`*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

❖ **Insert**: If bucket is full, *split* it (*allocate new page, re-distribute*).

❖ *If necessary*, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

Three basic alternatives for data entries in any index

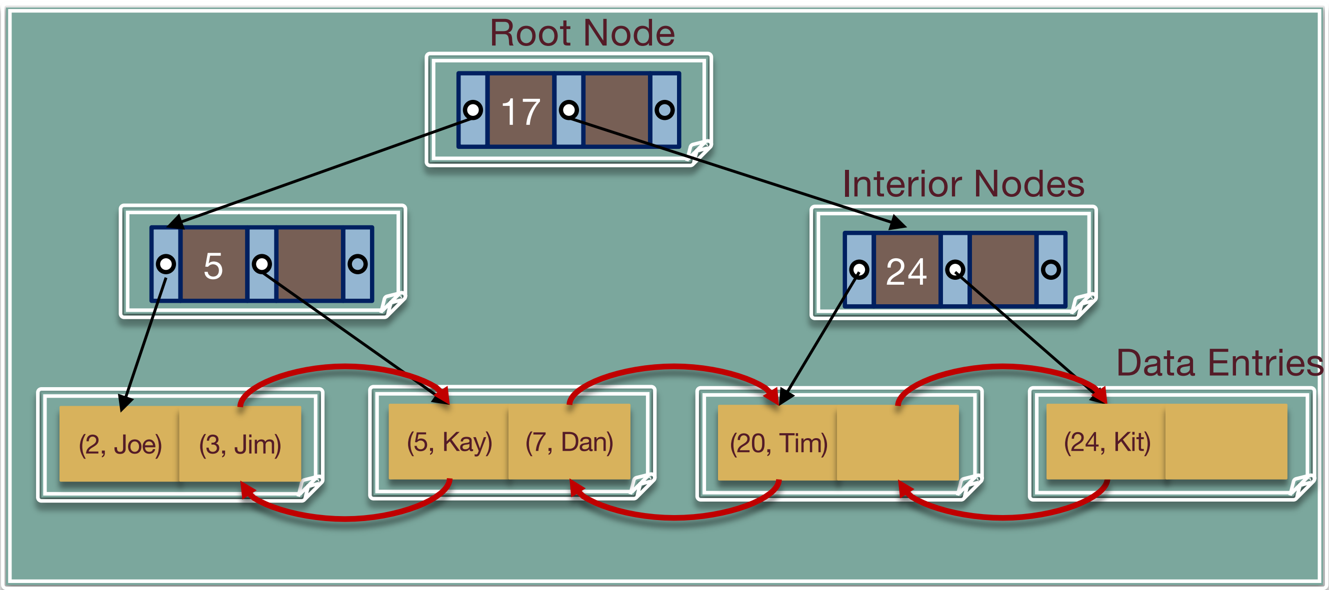
- Three basic alternatives for data entries in any index
  - Alternative 1: By Value
  - Alternative 2: By Reference
  - Alternative 3: By List of references

Credit: J. Hellerstein

Alternative 1 Index (B+ Tree)

- Record contents are stored in the index file
  - No need to follow pointers

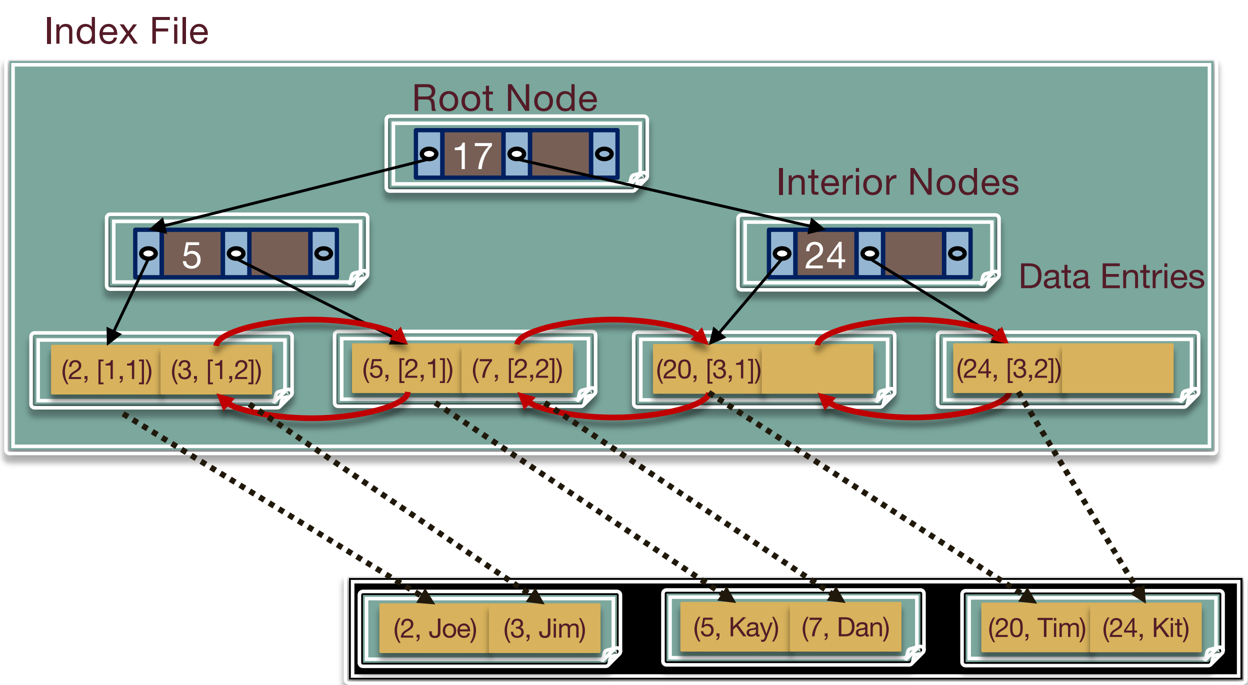
uid	name
2	Joe
3	Jim
5	Kay
7	Dan
20	Tim
24	Kit



Alternative 2 Index

uid	name
2	Joe
3	Jim
5	Kay
7	Dan
20	Tim
24	Kit

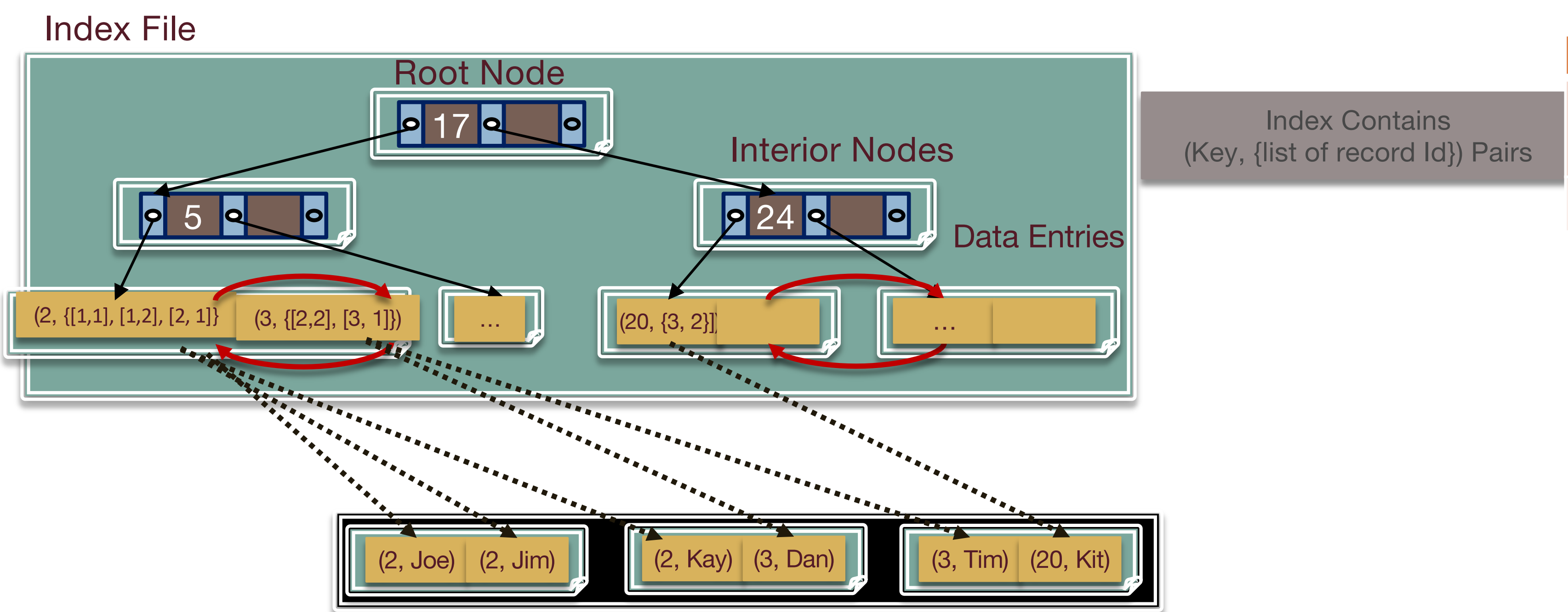
- Alternative 2: **By Reference, <k,** rid of matching data record>



Index Contains (Key, Record Id) Pairs

Alternative 3 Index

- Alternative 3: **By List of references**, <**k**, list of rids of matching data records>
  - Alternative 3 more compact than alternative 2
    - For very large rid lists, single data entry spans multiple blocks



Key	Record Id
2	{[1,1], [1,2], [1,3]}
3	4

Indexing By Reference

- Both Alternative 2 and Alternative 3 index data *by reference*
- By-reference is *required* to support multiple indexes per table
  - Otherwise, we would be replicating entire tuples
  - Replicating data leads to complexity when we're doing updates, so it's something we want to avoid

Alternative 2 vs Alternative 3 Table Illustration

Alternative 2
Index data entries

Key	Record Id
Gonzalez	[3, 1]
Gonzalez	[3, 2]
Gonzalez	[3, 3]
Hong	[3, 4]

SSN	Last Name	First Name	Salary
123	Gonzalez	Amanda	$400
443	Gonzalez	Joey	$300
244	Gonzalez	Jose	$140
134	Hong	Sue	$400

Alternative 3
Index data entries

Key	Record Id
Gonzalez	[3, {1, 2, 3}]
Hong	[3,4]