# Systems Design and Security

UNIFIED MODELING LANGUAGE™

## Part 8: Query Processing

http://staffwww.dcs.shef.ac.uk/people/A.Simons/

Home ⇒ Teaching ⇒ Lectures ⇒ COM2008/COM3008

The University Of Sheffield.

# Bibliography

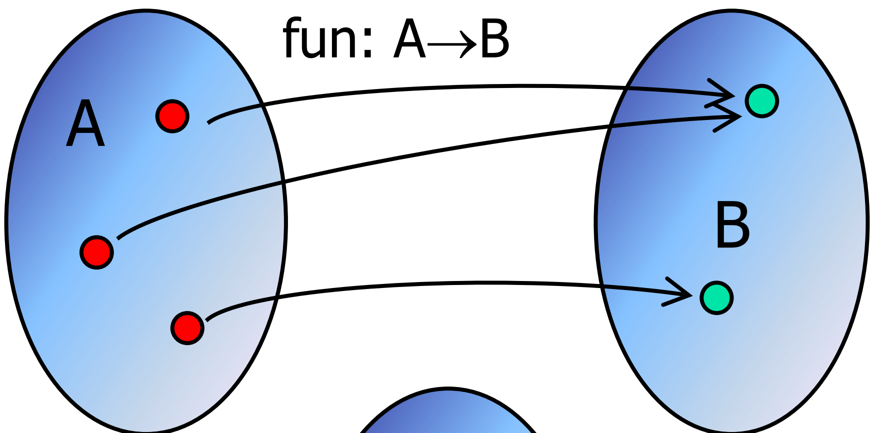
- Database Systems
  - T Connolly and C Begg, Database Systems – a Practical Approach to Design, Implementation and Management, 6th ed., Pearson, 2014.
  - C J Date, An Introduction to Database Systems, 8th ed., Pearson, 2003.
- Java JDBC and MySQL
  - http://download.oracle.com/javase/6/docs/api/java/sql/package-summary.html
  - http://www.mysql.com/downloads/

# Outline

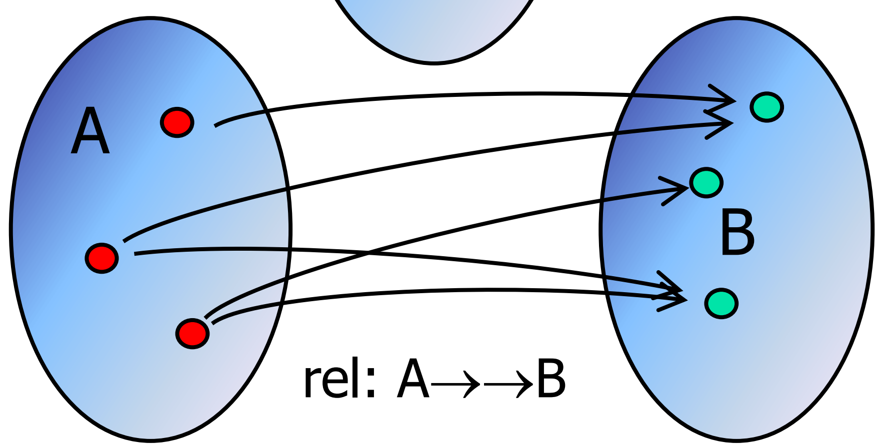
- Mathematical relations
- Relational algebra and SQL
- Data definition and manipulation
- Queries and aggregate functions
- Optimizing and indexing queries
- Java JDBC API to the MySQL database

Reading: Date chapters 4, 6, 7, 17, appendices A, B;
Connolly and Begg, chapters 4, 5, 6, 7, 23

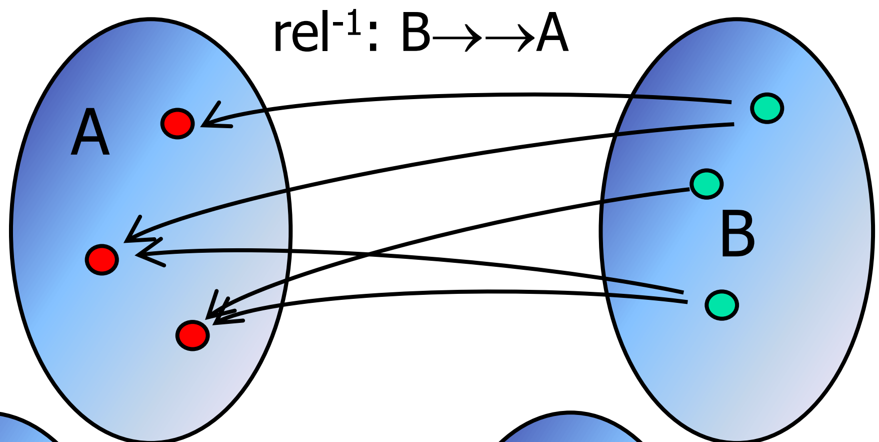
# Mathematical Relations



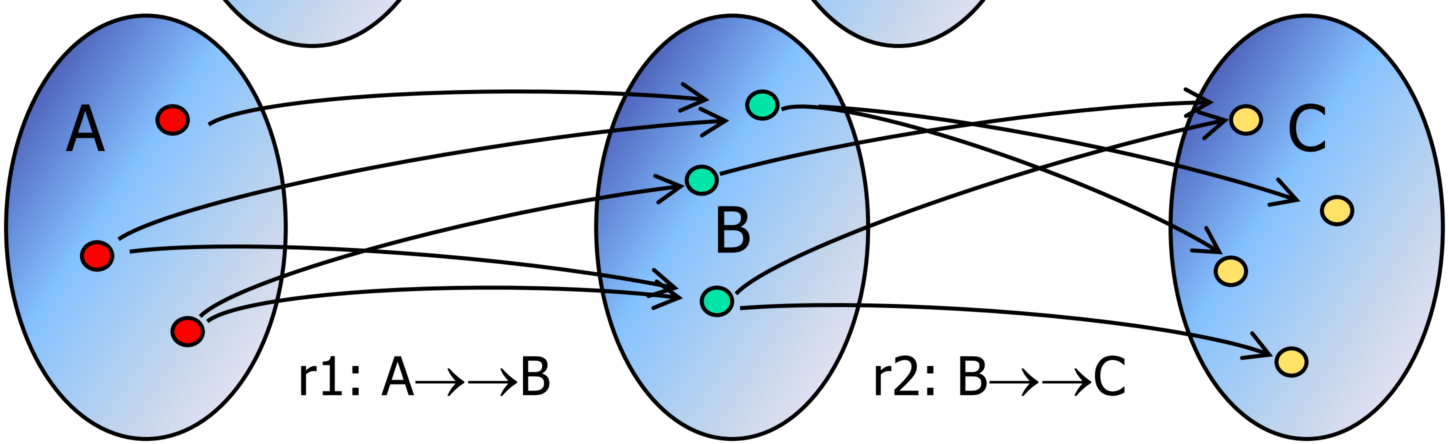

- Function is M:1
  - eg: square(int) : int
- Relation is M:N
  - eg: sqroot(int) : int



# Inverse and Join




- Inverse relation
  - reverse the arrows
- Joined relation
  - join two relations on common elements



# Relational Algebra

- Definition of a relation
  - Cartesian product: $A \times B \times C = \{(a, b, c) \mid a \in A, b \in B, c \in C\}$
  - a relation is a subset: $r(A, B, C) \subseteq A \times B \times C$
  - n-place relation, a set of n-tuples (a, b, c, ...)
  - bidirectional: $r(A, B) = r: A \rightarrow\rightarrow B \cup r^{-1}: B \rightarrow\rightarrow A$
- Relational Algebra (Codd, 1970)
  - set operations – union, intersection, difference
  - product: create a "longer" relation combining two others
  - select: filter the tuples of a relation, using a predicate
  - project: create a "shorter" relation, with fewer places
  - join: similar to product, but merging one or more places

# Select

- Sales

| orderID | city | productID | quantity |
|---|---|---|---|
| S1 | London | P1 | 100 |
| S1 | London | P2 | 100 |
| S2 | Paris | P1 | 200 |
| S2 | Paris | P2 | 200 |
| S3 | Paris | P2 | 300 |

- SELECT city = London (Sales)

filters a subset of the relation

| orderID | city | productID | quantity |
|---|---|---|---|
| S1 | London | P1 | 100 |
| S1 | London | P2 | 100 |

# Project

- Sales

| orderID | city | productID | quantity |
|---|---|---|---|
| S1 | London | P1 | 100 |
| S1 | London | P2 | 100 |
| S2 | Paris | P1 | 200 |
| S2 | Paris | P2 | 200 |
| S3 | Paris | P2 | 300 |

- PROJECT city, productID (Sales)

| city | productID |
|---|---|
| London | P1 |
| London | P2 |
| Paris | P1 |
| Paris | P2 |

result has no defined primary key columns (cannot tell which)

duplicate tuples are merged in the result – since it is a set

# Product (Cross-Join)

- Cities

| city |
|---|
| London |
| Paris |

Quantities

| productID | quantity |
|---|---|
| P1 | 100 |
| P2 | 200 |

- Cities PRODUCT Quantities

the combination of all cities...

| city | productID | quantity |
|---|---|---|
| London | P1 | 100 |
| London | P2 | 200 |
| Paris | P1 | 100 |
| Paris | P2 | 200 |

always creates a compound primary key

...with all quantities

# Natural Join

- Orders

| city | productID |
|---|---|
| London | P1 |
| Paris | P2 |
| London | P2 |
| Paris | P4 |

Quantities

| productID | quantity |
|---|---|
| P1 | 100 |
| P2 | 200 |
| P3 | 100 |

- Orders JOIN Quantities

the merger of all orders with all quantities...

...joined on the common column

| city | productID | quantity |
|---|---|---|
| London | P1 | 100 |
| London | P2 | 200 |
| Paris | P2 | 200 |

omitting rows for which there was no match in the other table

# Join Varieties

- Natural join
  - joins tables on same-named columns (one copy in result)
- Inner join
  - joins on explicitly named columns (all in the result)
- Left outer join
  - also includes LH rows with no match in the RH table (nulls!)
- Right outer join
  - also includes RH rows with no match in the LH table (nulls!)
- Full outer join
  - the union of left, right outer joins – nulls on both sides

# Left Outer Join

- Orders

| city | productID |
|---|---|
| London | P1 |
| Paris | P2 |
| London | P2 |
| Paris | P4 |

Quantities

| productID | quantity |
|---|---|
| P1 | 100 |
| P2 | 200 |
| P3 | 100 |

all rows from left table in the result

no matching row in right table

- Orders LEFT OUTER JOIN Quantities

| city | Ord.prodID | Qty.prodID | quantity |
|---|---|---|---|
| London | P1 | P1 | 100 |
| London | P2 | P2 | 200 |
| Paris | P2 | P2 | 200 |
| Paris | P4 | null | null |

# Lab 1: Algebra Exercise

Run a Poll

- Sales

| orderID | city | productID | quantity |
|---|---|---|---|
| S1 | London | P1 | 100 |
| S1 | London | P2 | 100 |
| S2 | Paris | P1 | 200 |
| S2 | Paris | P2 | 200 |
| S3 | Paris | P2 | 300 |

- Write the expression that yields a table containing orders with their corresponding cities and quantities – what is the result?
- Write an expression that yields a table of products and quantities, with greater than 100 of each product in the result
- Define a table Country linking each city to a country in {France, UK}, then write an expression identifying the country for each Sales order

# SQL

- SQL – the standard DB query language
  - name originally from “Structured Query Language”
  - 1987 international standard merely refers to “SQL”
  - later versions SQL2, SQL3 support richer types
- Differences from relational algebra
  - based on relational algebra and relational calculus
  - syntax for defining, manipulating and querying tables
  - queries return lists, not sets! (Codd hates this)
  - some poorly-chosen operation names!
    - eg: SQL “SELECT” does a relational algebra PROJECT
    - eg: SQL “WHERE” does a relational algebra SELECT


© Matt Groening

The University Of Sheffield.

# Data Definition

- CREATE TABLE defines tables
- Identify primary, foreign keys

BOOL, BOOLEAN (synonyms)
INT, INTEGER (likewise)
CHAR, CHARACTER (likewise)

```
CREATE TABLE BookTitle (
  isbn INT (13) NOT NULL PRIMARY KEY,
  name VARCHAR (30),
  author VARCHAR (20));

CREATE TABLE BookCopy (
  copyID CHAR (30) NOT NULL PRIMARY KEY,
  shortLoan BOOL DEFAULT FALSE,
  isbn INTEGER (13),
  FOREIGN KEY (isbn) REFERENCES BookTitle);
```

identifies primary key, which must not be null

CHAR is padded with 0s
VARCHAR is not padded

BOOL synonym for TINYINT(1)
integers have a field-width

relates foreign key to column in other table

# Linker Definition

- CREATE TABLE also defines linkers
- Identify compound primary, foreign keys

```
CREATE TABLE Loan (
  memberID INT (10) NOT NULL,
  copyID CHAR (30) NOT NULL,
  issueDate DATE,
  dueDate DATE,
  PRIMARY KEY (memberID, copyID),
  FOREIGN KEY (memberID) REFERENCES Borrower,
  FOREIGN KEY (copyID) REFERENCES BookCopy (copyID));
```

alternative style identifies the primary key after defining all the columns

DATE is a primitive type, format is: YYYY-MM-DD

identifies both foreign keys; can refer to a column having the same name (first example), or to a column with a different name (second example)

# Data Manipulation

- Commands include: INSERT, UPDATE, DELETE

```
INSERT INTO BookTitle VALUES
  (0747532699, 'Harry Potter and the Philosopher\'s Stone',
'J K Rowling'),
  (0552153370, 'Unseen Academicals', 'Terry Pratchett'),
  (0316160172, 'Twilight', 'Stephanie Meyer');
```

comma-separated list of rows, care with strings and quotes

```
UPDATE BookCopy SET shortLoan = TRUE
  WHERE copyID = '823.452.767.2';
```

certain BookCopy instances are reserved for short loan

```
DELETE FROM Loan
  WHERE copyID = '823.452.767.2';
```

this BookCopy is now no longer on loan to anybody

# Simple Query

- SQL command is: SELECT <cols> WHERE <pred>
- relational algebra: PROJECT <cols> (SELECT <pred> (rel))

```
SELECT name, author FROM BookTitle
  WHERE isbn = 0747532699;
```

isbn identifies a single row, returns a table with the title and the author

```
SELECT name, author FROM BookTitle
  WHERE name LIKE 'Harry Potter%';
```

predicate matches many titles, returns a table with many titles and corresponding authors

```
SELECT name, author FROM BookTitle
  WHERE name IN ('Twilight', 'New Moon')
    ORDER BY author, name;
```

similar, based on title being in the given set of names, with results in alphabetical order of author, then book name

# Lists and Sets

- SQL command SELECT returns a list
- SQL command SELECT DISTINCT returns a set

```
SELECT author FROM BookTitle
  WHERE title LIKE 'Harry Potter%';
```

predicate matches many titles, returns a table in which author 'J K Rowling' appears many times!

```
SELECT DISTINCT author FROM BookTitle
  WHERE title LIKE 'Harry Potter%';
```

now she only appears once

```
SELECT DISTINCT author FROM BookTitle;
```

projects out the set of authors

```
SELECT * FROM BookTitle;
```

* wildcard, selects all columns

# Cross Join

- SQL has explicit CROSS JOIN command (ie product)
- SQL computes cross-joins implicitly when many tables are referenced, with no restriction

```
SELECT * FROM BookTitle CROSS JOIN BookCopy;
```

cross-join or product, explicit syntax, all combinations

```
SELECT * FROM BookTitle, BookCopy;
```

cross-join or product, implicit syntax, all combinations

| <u>bt.isbn</u> | bt.name | bt.author | <u>bc.copyID</u> | <u>bc.i</u>sbn | bc.shortLoan |
|---|---|---|---|---|---|
| 0747532699 | Harry Potter.. | J K Rowling | 823.452.767.2 | 0747532699 | FALSE |
| 0552153370 | Unseen Aca.. | Terry Pratc.. | 823.452.767.2 | 0747532699 | FALSE |
| ... | ... | ... | 823.452.767.2 | 0747532699 | FALSE |

# Inner Join

- SQL has explicit INNER JOIN ... ON command
- SQL computes inner joins implicitly when many tables are referenced and a WHERE-clause restricts columns

```
SELECT * FROM BookTitle INNER JOIN BookCopy
  ON BookTitle.isbn = BookCopy.isbn;
```

inner join, explicit syntax; isbn column is repeated

```
SELECT * FROM BookTitle, BookCopy
  WHERE BookTitle.isbn = BookCopy.isbn;
```

inner join, implicit syntax; isbn column is repeated

| bt.isbn | bt.name | bt.author | bc.copyID | bc.isbn | bc.shortLoan |
|---|---|---|---|---|---|
| 0747532699 | Harry Potter.. | J K Rowling | 823.452.767.2 | 0747532699 | FALSE |
| 0552153370 | Unseen Aca.. | Terry Pratc.. | 823.452.723.3 | 0552153370 | TRUE |
| ... | ... | ... | ... | ... | ... |

# Natural Join

- SQL has explicit NATURAL JOIN command
- SQL computes natural joins implicitly when many tables referenced and a USING restriction is applied

```
SELECT * FROM BookTitle NATURAL JOIN BookCopy;
```

natural join, explicit syntax; single isbn column

```
SELECT * FROM BookTitle, BookCopy
  USING (isbn);
```

natural join, implicit syntax; single isbn column

shared column appears first, not qualified by table name

| isbn | bt.name | bt.author | bc.copyID | bc.shortLoan |
|---|---|---|---|---|
| 0747532699 | Harry Potter.. | J K Rowling | 823.452.767.2 | FALSE |
| 0552153370 | Unseen Aca.. | Terry Pratc.. | 823.452.723.3 | TRUE |
| … | … | … | … | … |

# Typical Query

- Connect tables using inner join (or natural join)
- Project out the desired columns as a new table

```
SELECT name, author, copyID FROM BookTitle, BookCopy
  WHERE BookTitle.isbn = BookCopy.isbn;
```

qualify with table names

```
SELECT name, author, copyID FROM BookTitle bt, BookCopy bc
  WHERE bt.isbn = bc.isbn;
```

...or using alias names

result is a new table

| bt.name | bt.author | bc.copyID |
|---|---|---|
| Harry Potter.. | J K Rowling | 823.452.767.2 |
| Unseen Aca.. | Terry Pratc.. | 823.452.723.3 |
| ... | ... | ... |

# Alias Names

- SQL supports alias names for tables, select expressions
- Helps to disambiguate columns that have identical names
- The keyword AS is usually optional

```
SELECT bt.name, bt.author, bc.copyID
        FROM BookTitle AS bt, BookCopy AS bc
 WHERE  bt.isbn = bc.isbn;
```

Explicit aliases for BookTitle, BookCopy

```
SELECT bt.name, bt.author, bc.copyID
        FROM BookTitle bt, BookCopy bc
 WHERE bt.isbn = bc.isbn;
```

Implicit alias names for the same tables

```
(SELECT * FROM Loan WHERE dueDate < ‘2022-09-20’) AS overdue
```

Alias name for the result of a SELECT

# Complex Query

- Join multiple tables, project arbitrary columns
- Use compound conditions in the WHERE-clause

```
SELECT b.forename, b.surname, t.name, m.dueDate
  FROM Borrower b, Loan m, BookCopy c, BookTitle t
  WHERE m.memberID = b.memberID        joins Loan, Borrower
    AND m.copyID = c.copyID            joins Loan, Copy
    AND c.isbn = t.isbn                joins Copy, Title
    AND m.dueDate < '2022-09-20';      overdue after 20th September
```

| b.forename | b.surname | t.name | m.dueDate |
|---|---|---|---|
| John | Smith | Harry Potter... | 2022-09-03 |
| Jane | Doe | Twighlight | 2022-08-25 |
| ... | ... | ... | ... |

predicate may use any comparison operator, or IN a set of values, etc.

# Aggregate Functions

- Non-relational part of SQL
  - "summarising" sets of data to yield statistics
  - compute set $\rightarrow$ scalar, like reduce in functional languages
  - examples:  EXISTS, COUNT, SUM, MAX, MIN, AVG, ...

```
SELECT COUNT (*) FROM Loan;
```

total number of Loans

```
SELECT COUNT (DISTINCT memberID)
  FROM Loan;
```

total number of Borrowers with at least one Loan

```
SELECT t.name FROM BookTitle t, BookCopy c
  WHERE EXISTS (SELECT * FROM Loan m
    WHERE m.copyID = c.copyID)
      AND t.isbn = c.isbn;
```

the name of any book that is currently on loan

# Grouping Statistics

- SQL command GROUP BY splits up statistics by groups
- SQL predicate HAVING is like WHERE, but for groups
- Can eliminate summary columns from main tables

SELECT copyID FROM Loan;

all the book copies that are currently on loan

SELECT memberID, COUNT (*) AS loans
  FROM Loan GROUP BY memberID;

a table containing all unique borrowers and a count of their current loans

SELECT memberID, COUNT (*) AS loans
FROM Loan GROUP BY memberID
    HAVING COUNT (*) > 3;

likewise, but only if the count of loans is greater than 3

| memberID | loans |
|---|---|
| 1012234 | 5 |
| 1667753 | 4 |
| 1784532 | 6 |

# Usefulness

- Aggregate functions very useful
  - summarising data is what users seem to want
  - probably why SQL has survived so long

```
SELECT t.name, MIN(m.issueDate) FROM BookTitle t, BookCopy c, Loan m
  WHERE t.isbn = c.isbn AND c.copyID= m.copyID
    GROUP BY t.name
```

the earliest a book was issued

```
SELECT t.name, COUNT (*) AS out FROM BookTitle t, BookCopy c, Loan m
  WHERE t.isbn = c.isbn AND c.copyID= m.copyID
    AND m.issueDate BETWEEN '2009-01-01' AND '2009-12-31'
      GROUP BY t.name
```

the number of times each book was issued in 2009

# Lab 2: SQL Exercise

Run a Poll

- Write SQL data definition/manipulation expressions
  - to create the Borrower table (like slide 16)
  - Borrowers have a memberID, forename and surname
  - to populate this table with some borrowers (like slide 17)

- Write SQL query/manipulation expressions
  - to find the unique titles of all books borrowed by anyone with the surname 'Smith' before 2022-10-01
  - to issue a loan to John Smith, of one copy of "Harry Potter and the Philosopher's Stone"

# Query Optimization

- Efficiency issues
  - how many rows are there to find (time)?
  - how difficult is it to find them (#joins)?
  - where to store intermediate results (space)?
- Possible approaches
  - structure queries using subqueries (the 'S' in 'SQL')
  - filter large datasets first, before joining (select subsets)
  - project columns, before joining (project shorter tables)
- Database Management Systems
  - commercial DBMSs use automatic query optimization
  - reorganise the order in which parts of query are executed
  - DBMS must know in advance what queries will be asked

# Table as a Scalar

- A common subquery returns a single-column table
- The result can be treated like a scalar set (or list)
- SQL has the IN set membership predicate

```
SELECT forename, surname FROM Borrower
  WHERE memberID IN (SELECT memberID FROM Loan
    WHERE dueDate >= '2022-10-01');
```

single-column table
used like a set

- This is an efficient kind of query optimisation
- Filters a large set from Loan before joining to Borrower

# Nested Subquery

- Sometimes more efficient to filter large datasets first
- Wrap the expensive query around the cheap filter query

```
SELECT b.forename, b.surname, t.name, m.dueDate
  FROM Borrower b, BookCopy c, BookTitle t,
    (SELECT * FROM Loan WHERE dueDate < '2022-09-20') AS od
  WHERE od.memberID = b.memberID
    AND od.copyID = c.copyID
    AND c.isbn = t.isbn;
```

computes a small subset of all Loans first; and returns a table, so can be joined

| b.forename | b.surname | t.name | od.dueDate |
|---|---|---|---|
| John | Smith | Harry Potter... | 2022-09-03 |
| Jane | Doe | Twighlight | 2022-08-25 |
| ... | ... | ... | ... |

# Data Indexing

- Storage and searching issues
  - scanning through data files is very expensive
  - use fixed-length records (rows) in flat files – why?
  - if an index refers to the row-position in a large file
  - can offer faster random access to a row by its index
- Handling very large data sets
  - what if the table is too big to fit onto one storage device?
  - what if the query only needs a few rows out of millions?
  - what if queries sort the same data in different orders?
  - need a better indexing system, typically a B-tree
  - B-tree is a memory graph that maps PKs to row indices

# B-Trees

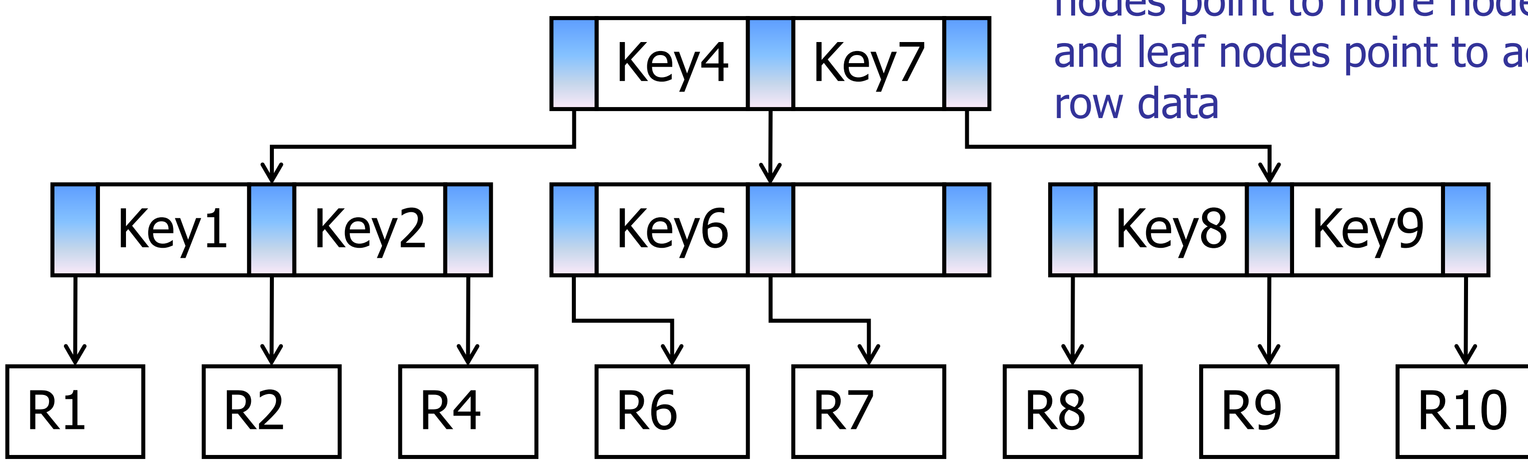
- Node representation

consists of alternate pointers and primary keys, ending with pointer

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| p1 | Key1 | p2 | Key2 | p3 | Key3 | ••• | pn | Keyn | pn+1 |

- Tree representation

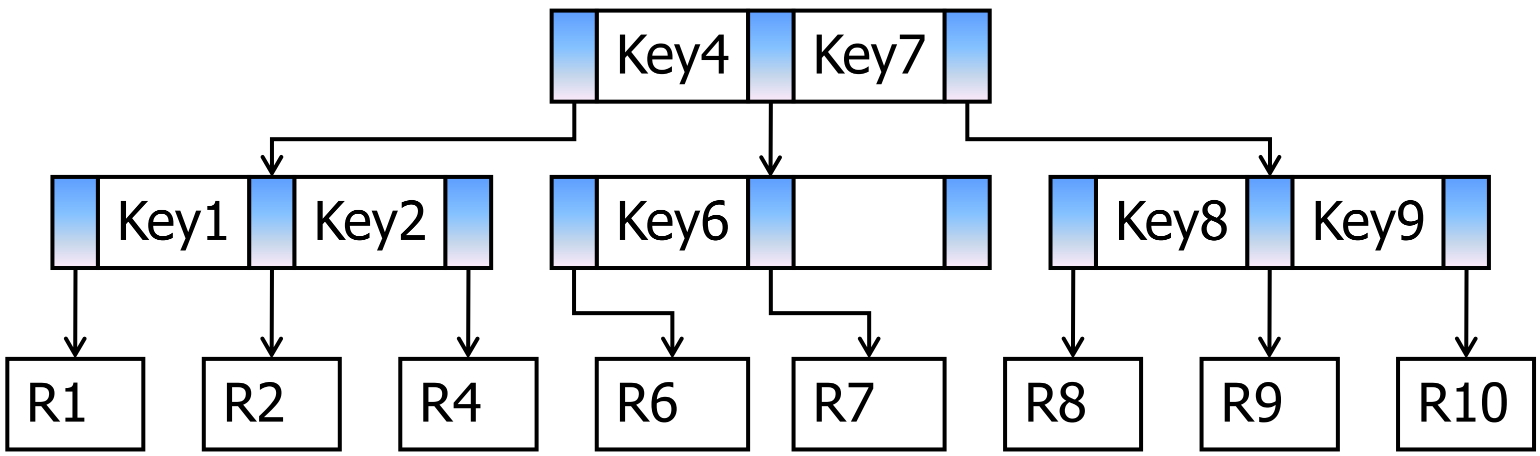
kind of tree, where branch nodes point to more nodes and leaf nodes point to actual row data



# B-Tree Properties

B+: only leaf-nodes point to actual data

- Balanced tree (a $B^+$-tree)
  - no node is less than half-full of pointers; keys are ordered
  - pointer Pi leads to all data rows with keys <= Key i
  - pointer Pi leads to all data rows with keys > Key i-1
  - last pointer Pn+1 leads to all data rows with keys > Key n

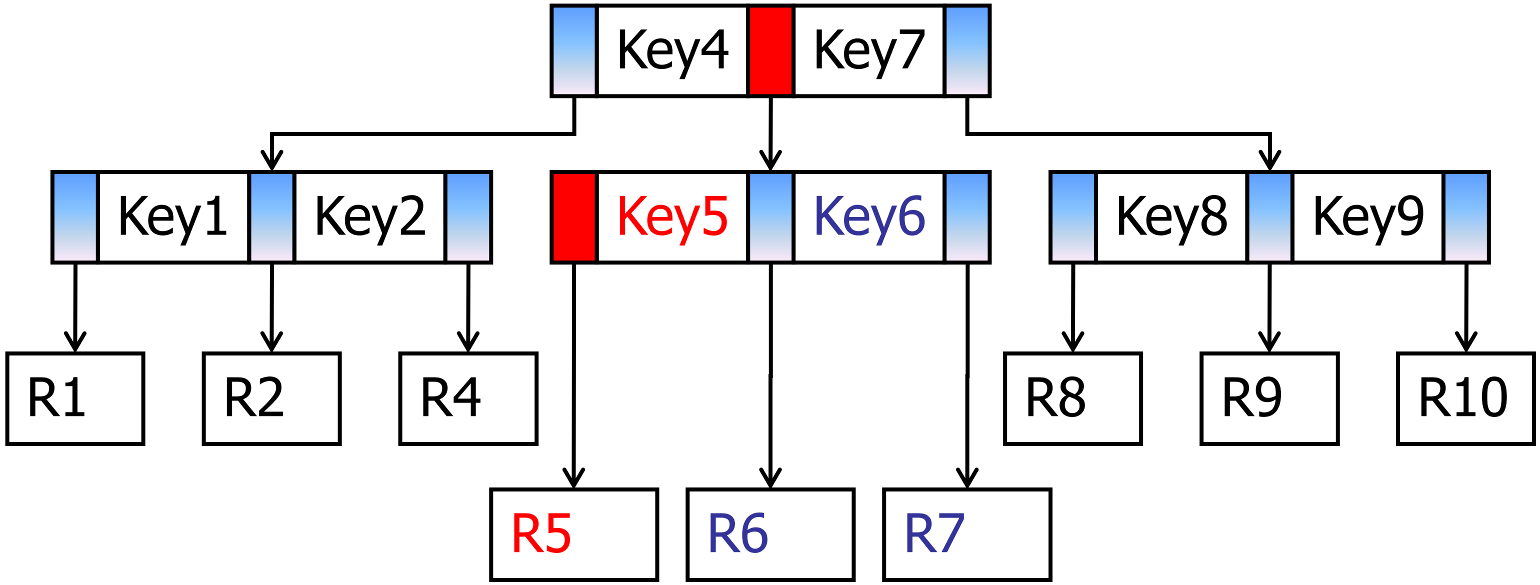


# Search Properties

- Access to data is fast
  - balanced: means all paths to data are of the same length
  - scalable: tree height is $\log_b(n)$ where #data=n, #branches=b
  - B-tree grows slowly compared to size of dataset
- Insert modifies few nodes
  - either: add a new pointer+key in a (half-full) leaf node
  - or: add a new leaf node, split/rearrange higher nodes
  - may repeat, ripple up the tree, add a new root
- Delete modifies few nodes
  - either: remove key+pointer from a (full) leaf node
  - or: delete (half full) leaf node and merge/rearrange higher nodes
  - may repeat, ripple up the tree, delete root, merge/child is new root

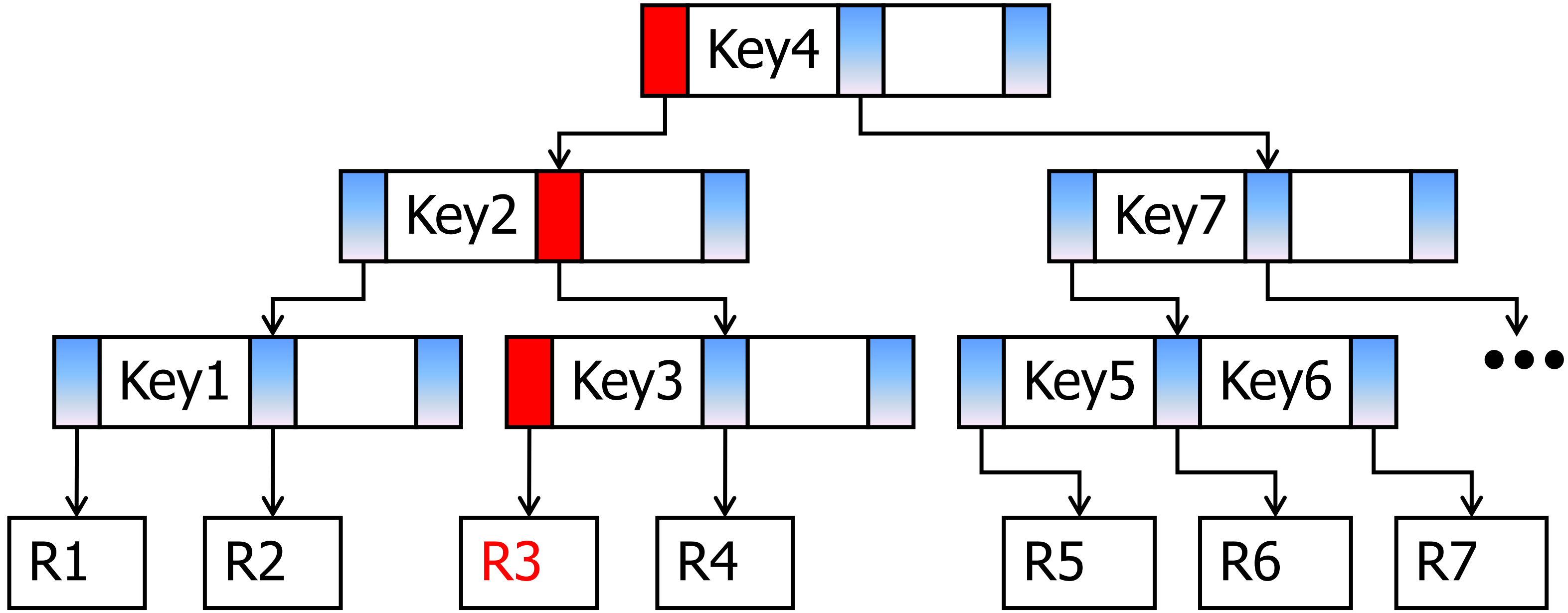
# Insert – free space

- Insert row with K5 as its primary key
  - must be on path from P7 (leads to all rows > K4 and <= K7)
  - node with K6 has space, so shift up and add K5+P5



# Insert – split nodes

- Insert row with K3 as its primary key
  - no space in node K1-K2, so split node and add new node
  - no space in root K4-K7, so split root and add new root



# Summary

- Relations are n-place, bi-directionally linked maps, a subset of the Cartesian product of n base types
- Relational algebra select, project and join operations can be nested to describe all kinds of data searching
- SQL is an impure implementation of relational calculus, with a data definition and data manipulation language
- SQL "select" is actually a "project" operation; SQL "where" is actually a "select" operation in relational algebra!
- Aggregate functions in SQL are very useful in practice and allow computation of dependent values – minimizes storage needs
- Query optimisation determines best execution order; while B-trees index large data sets and provide fast access to rows