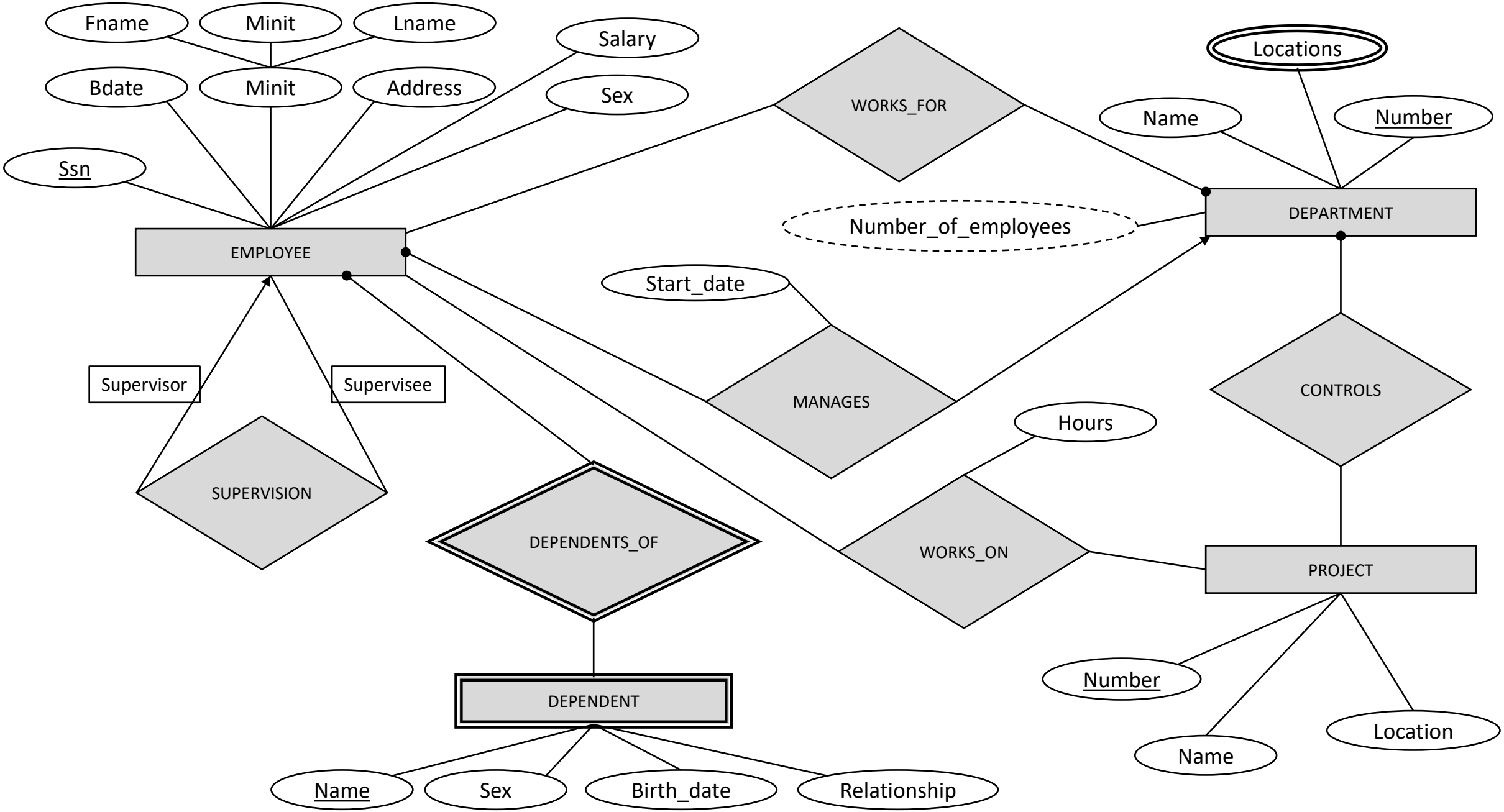
# TÖL303G

# Gagnasafnsfræði

Snorri Agnarsson

# Einindavenslagagnalíkön


Fname
Minit
Lname
Salary
Bdate
Minit
Address
Sex
Ssn
WORKS_FOR
Locations
Name
Number
DEPARTMENT
Number_of_employees
EMPLOYEE
Start_date
MANAGES
CONTROLS
Supervisor
Supervisee
Hours
SUPERVISION
DEPENDENTS_OF
WORKS_ON
PROJECT
DEPENDENT
Number
Name
Location
Name
Sex
Birth_date
Relationship

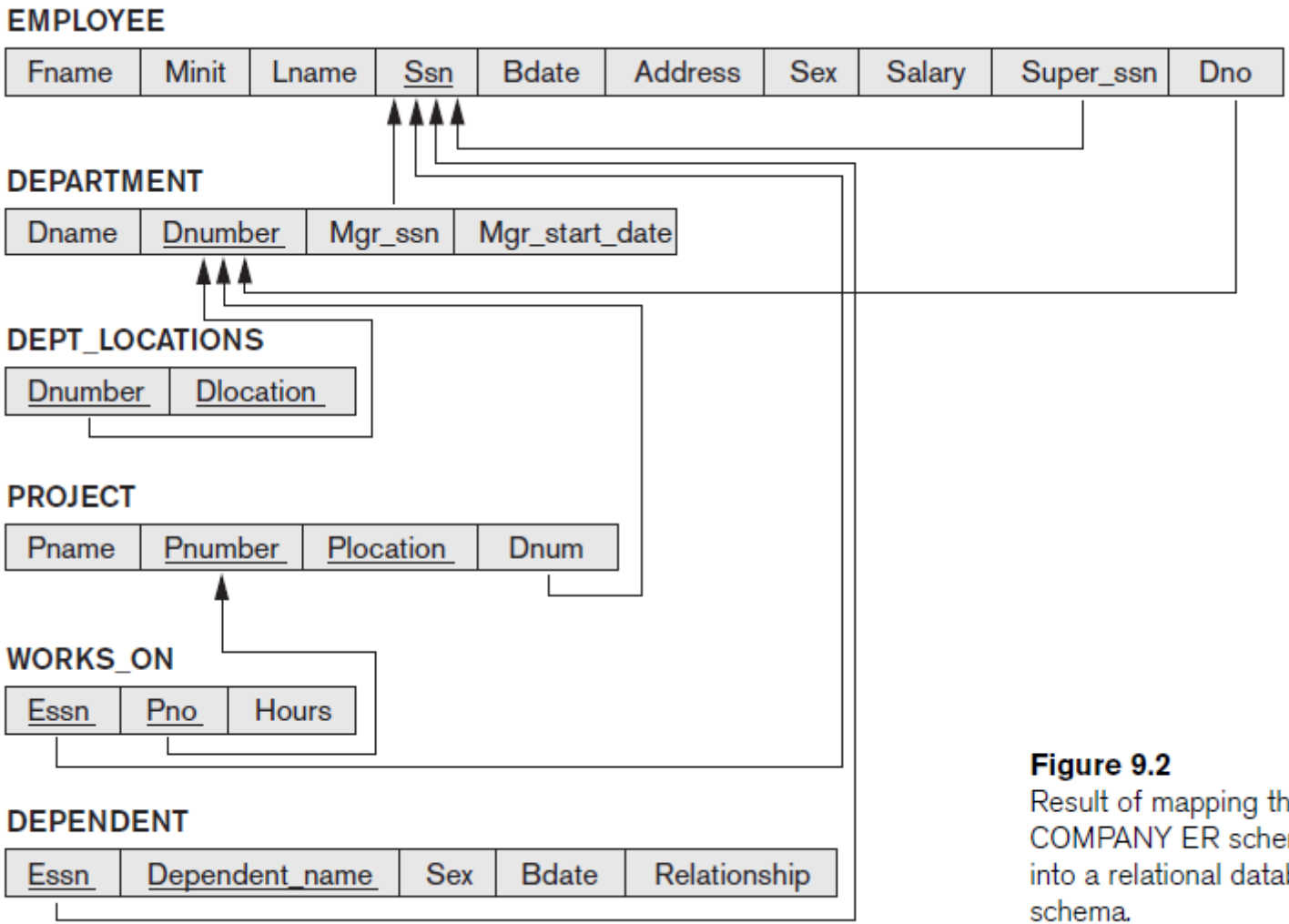



**Figure 9.2**
Result of mapping the COMPANY ER schema into a relational database schema.

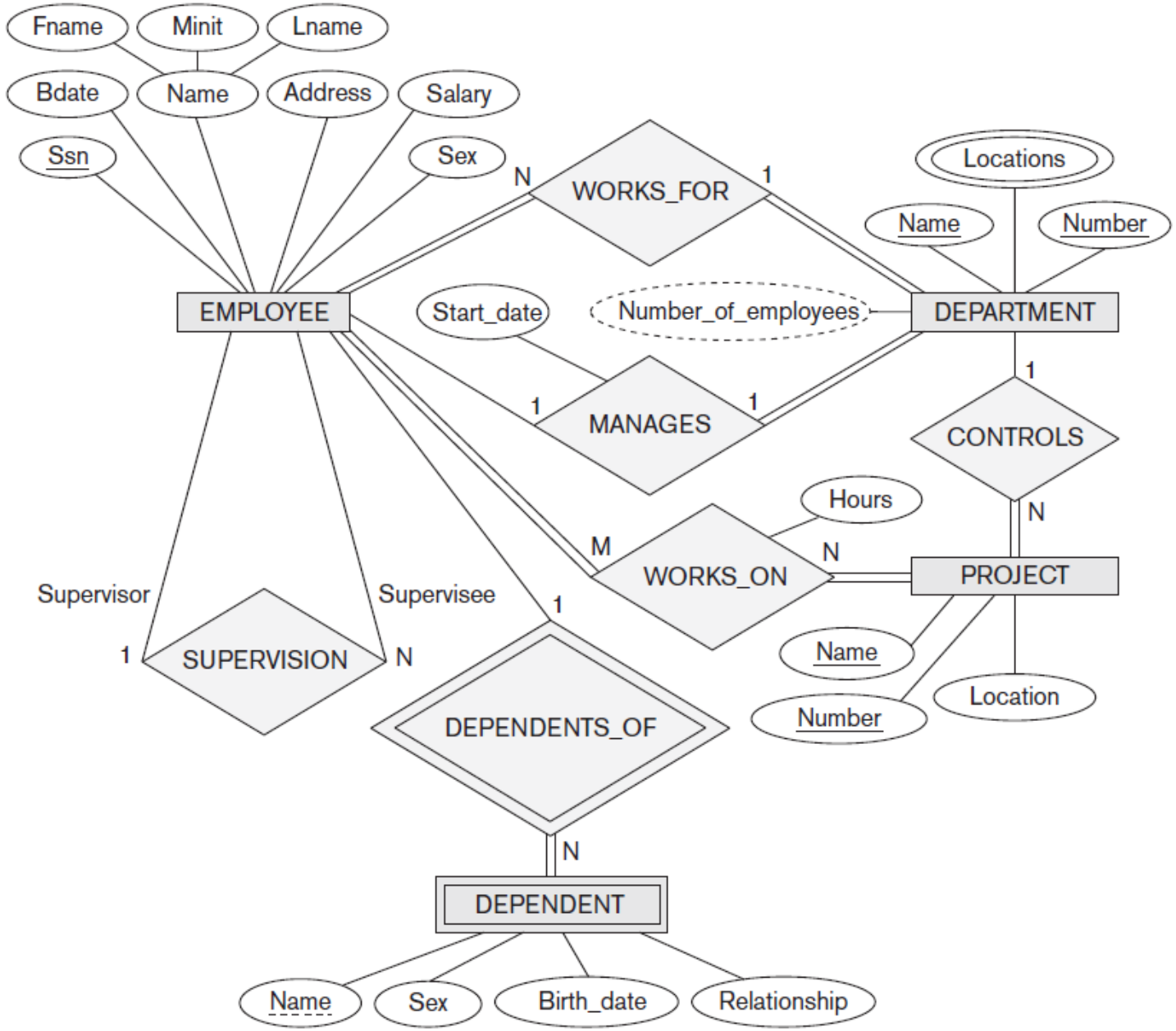



**Figure 7.2**
An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter and is summarized in Figure 7.14.

# Sýnir og vísar
# Views and Indexes

# Sýn (View)

- Í SQL er hægt að gefa algengum fyrirspurnum nafn með CREATE VIEW
  In SQL we can name common queries using CREATE VIEW
- Þá verður til sýn (view) á gagnagrunninn sem nota má sem töflunafn í öðrum fyrirspurnum
  This creates a view on the database that can be used as a table name in other queries
- Sýnir einfalda flóknar fyrirspurnir og gera oft copy/paste óþarfi
  Views simplify complicated queries and often make copy/paste unnecessary

```
CREATE VIEW ParamountMovie AS
SELECT title,year
FROM Movie
WHERE studioName='Paramount';
```

# Sýn (View)

- Við getum notað sýnina ParamountMovie eins og hverja aðra töflu
  We can use the view ParamountMovie just like any other table

```
SELECT title
FROM ParamountMovie
WHERE year > 1980;
```

- Þótt ParamountMovie sé hluti af Movie töflunni getum við aðeins vísað í þá dálka sem skilgreindir eru í sýninni
  Even though ParamountMovie is part of the Movie table we can only refer to the columns defined in the view

# WITH

- Sýnin sem við skilgreinum verður hluti af gagnagrunninum og kemur fram í schemanu
  The view created becomes part of the database and will be shown in the schema
- Ef við viljum einfalda flókna fyrirspurn en viljum ekki vista nýja sýn í gagnagrunninum þá getum við notað CREATE TEMPORARY VIEW til að búa til sýn sem hverfur þegar tengingunni á gagnagrunninn er lokað
  We can also create temporary views that disappear when the connection to the database is closed
- **Einnig má nota WITH/We can also use WITH:**

```
WITH M AS (SELECT model,price FROM Laptop
           UNION SELECT model,price FROM Pc
           UNION SELECT model,price FROM Printer)
SELECT model FROM M
WHERE price=(SELECT MAX(price) FROM M);
```

# Sýnir og breytingar – Views and Updates

- Það er hægt að breyta sýn undir vissum kringumstæðum
  Sometimes updates can be performed through views
  - Almennt er það ekki góð hugmynd – Generally not a good idea
  - Helsti tilgangur með sýn er – The main purpose of a view is
    - að leyfa takmarkaðan aðgang að gögnum og
      allow limited access to data and
    - að geyma algengar fyrirspurnir
      store common queries
- Með sýn er hægt að veita sumum notendum takmarkaðan aðgang að hluta gagnanna
  Views can allow some users limited access to part of the data
- Með sýn er einnig hægt að vera með flókið BCNF eða 3NF niðurbrot í margar töflur en sýna öll gögnin í einni töflu gegnum sýn
  With a view it is also possible to have a complicated BCNF or 3NF decomposition into many tables, but show all the data through a view

# Vísar (Index)

- SQL þarf ekki að skila niðurstöðum í ákveðinni röð nema tilgreint sé ORDER BY
  SQL does not have to return results in any given order unless specified using ORDER BY
  - Röðun er dýr, kostar $\mathrm{O}(n \log n)$ fyrir $n$ raðir
    Sorting is expensive, costs $\mathrm{O}(n \log n)$ for $n$ rows
- Þegar við gerum tengingar á töflum (join) þurfum við framkvæma leit eða samröðun (merge)
  When we join tables we need to perform searches or merges
- Til þess að leitir og samraðanir séu ódýrar er gott að sjá til þess að gögnin séu geymd og aðgengileg í réttri röð
  For searches and merges to be cheap it is good to ensure that the data are stored and accessible in the appropriate order

# INDEX

- Við búum til vísi (index) á töflu með
  Create an index to a table using

```
CREATE INDEX TitleIndex ON Movie(title);
```

- eða/or

```
CREATE INDEX KeyIndex ON Movie(title,year);
```

# Harðir Diskar – Hard Drives

- Eru hægir, mjög, mjög hægir – Are slow, very, very slow
  - Sjá/See: https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html
- Tölvunet eru ennþá hægari
  Computer networks are even slower

# B-tré – B-trees

- B-tré geyma marga lykla í hverjum hnúti í tré
  - Hver hnútur inniheldur $K$ lykla og vísar í $K + 1$ undirtré
  - Hnútarnir mega hafa mest $m$ lykla og minnst $m/2$ lykla (nema rótin)
  - Lyklarnir í trénu eru í milliröð (inorder röð), bæði innan hvers hnútar og undirtrén
  - Tréð er í fullkomnu jafnvægi, þ.e. sérhver leið frá rót til laufs er jafn löng
  - Hæðin á trénu verður $\log_m\left(\frac{n+1}{2}\right)$ fyrir $n$ lykla
  - Leit, innsetning og eyðing kosta hver um sig $O(\log n)$ þar sem $n$ er fjöldi gilda í trénu
- 2-3-4 tré eru afbrigði af B-trjám
  - Með $m = 4$
- Rauð-svört tvíleitartré eru náskyld 2-3-4 trjám

# B-tré – B-trees

- B-trees store many keys in each node in a tree
  - Each node contains $K$ keys and refers to $K + 1$ subtrees
  - The nodes can contain a maximum of $m$ keys and a minimum of $m/2$ keys (except the root)
  - The keys in the tree are sorted inorder, both within each node and in the subtrees
  - The tree is perfectly balanced, i.e. Each path from root to leaf is of the same length
  - The height of the tree becomes $\log_m \left(\frac{n+1}{2}\right)$ for $n$ keys
  - Searces, insertions and deletions each cost $O(\log n)$ where $n$ is the number of values in the tree
- 2-3-4 trees are a version of B-trees
  - With $m = 4$
- Red-black binary search trees are closely related to 2-3-4 trees

# Síður (pages)

- Diskar geyma gögn í síðum (pages)
  - Þegar við náum í gögn af diski eru gögnin sótt í heilum síðum
  - Algeng stærð síðu er 4K
  - Oft fer mestur tími í gagnavinnslu í að sækja síður af diski
- Með B-trjám lágmörkum við fjölda lestra
- Leitaraðferðin í B-trjám takmarkar fjölda lestraraðgerða
  - Nokkurs konar helmingunarleit
  - Þar með tökum við tillit til þess að diskalestur er mjög dýr aðgerð
  - Tvíleitartré væru verri kostur því fjöldi skrefa frá rót til laufs myndi aukast
- Sjá líka https://en.wikipedia.org/wiki/B-tree

# Pages

- Disks store data in pages
  - When data is retrieved from disk the data are fetched in whole pages
  - A common page size is 4K
  - Often most of the time in data processing is spent fetching data from disk
- With B-trees we minimize the number of fetches
- The search method of B-trees lowers the number of read operations
  - Related to binary search
  - Thereby we minimize the effect of the fact that disk reads are very expensive
  - Binary search trees would be slower because the number of steps from root to leaf would increase
- See also https://en.wikipedia.org/wiki/B-tree

# Ódýrar aðgerðir á B-tré

- Með B-trjám er ódýrt (tímaflækja $O(\log n)$ fyrir hverja færslu) að
  - Finna færslu með tilteknum lykli
  - Finna færsluna með minnsta lykil eða færsluna með minnsta lykil stærri en (eða $\geq$) tiltekið gildi
  - Finna færsluna með stærsta lykil eða færsluna með stærsta lykil minni en (eða $\leq$) tiltekið gildi
  - Bæta við færslu með tilteknum lykli á réttan stað miðað við röð lykla
  - Eyða færslu með tilteknum lykli
  - Ganga í gegnum færslur eða hluta af færslum í milliröð (inorder, lyklaröð) eða öfugri milliröð
    - Til dæmis er tiltölulega ódýrt að ganga í gegnum allar færslur með lyklum $x$ sem uppfylla $a \leq x \leq b$ fyrir einhver tiltekin gildi $a$ og $b$

# Inexpensive B-tree operations

- With B-trees it is inexpensive (time complexity $O(\log n)$ for each record) to
  - Find a record with a given key
  - Find the record with the smellest key or the record with the smallest key greater than (or $\geq$) a given value
  - Find the record with the greatest key or the record with the greatest key less than (or $\leq$) a given value
  - Add a record with a given key at the correct place considering the key order
  - Delete a record with a given key
  - Traverse records or a subset of records inorder or reverse inorder
    - For example it is relatively inexpensive to traverse all records with keys $x$ such that $a \leq x \leq b$ for some given values $a$ and $b$

# Tæting (hashing)

- Önnur aðferð til að búa til vísa er að nota tætingu (hashing)
- Þá er hverju gildi á lykli varpað í heiltölu með einhverri aðferð, sem vonandi gefur góða dreifingu
- Ef við erum ekki óheppin þá getur meðaltímaflækja til að leita að færslu með tilteknum lykli orðið $O(1)$
- Þessi aðferð er ekki eins sveigjanleg og B-tré
  - Gefur ekki kost á að ferðast gegnum færslur í lyklaröð og gefur ekki jafn mikið safn af ódýrum aðgerðum
  - Við gætum orðið óheppin, tímaflækjan er ekki endilega fyrirsjáanleg
  - Það er ekki endilega einfalt að finna góðar tætiaðferðir (hashing function)

# Hashing

- Another method of indexing is to use hashing
- Then each key value is mapped to an integer with some method that hopefully gives a good distribution
- If we are not unlucky the average time complexity of searching for a record with a given key can be $O(1)$
- This method is not as flexible as B-trees
  - Does not support traversing inorder and does not have as many inexpensive operations
  - We might be unlucky, the time complexity is not necessarily predictable
  - It is not always simple to find good hashing functions

# Vísar, kostir og gallar
# Indexes, the good and the bad

- Ef vísar hraða fyrirspurnum og tengingum af hverju búum við ekki til alla mögulega vísa?
  Why not create all conceivable indexes?
- Það kostar pláss að geyma vísinn – Costs space
- Þegar gögn eru uppfærð þarf að uppfæra vísinn, skrifa á disk – Costs time to update
- Sumar töflur eru svo litlar að það er einfaldara og fljótara að lesa þær allar í einu
  Some tables are too small to benefit from indexing

# Hvernig á að velja vísa?
# What to index on?

1. Ef við höfum aðallykil (PRIMARY KEY) sem verður notaður í tengingum (join) þá viljum við setja vísi á hann (SQLite gerir þetta alltaf fyrir okkur)
2. Ef við erum með algengar leitir, til dæmis leita að nemendum eftir nafni, þá setjum við vísi jafnvel þótt nafnið sé ekki lykill
3. Ef algengt er að beðið sé um gögn í röð eða leitað sé að bilum gagnagilda (dagsetningar í röð, heimilisföng í röð, nöfn í röð) þá setjum við vísi (B-tré eða samsvarandi)

# Hvernig á að velja vísa?
# What to index on?

1. If we have a PRIMARY KEY that will be used in joins then we want an index on it (SQLite always does this for us)
2. If we have common queries, for example searching for students by name, then we put an index even though the name is not a key
3. If it is common that data is asked for in a given order or if queries commonly search for sequences of data (dates in order, addresses in order, names in order) then we put an index (B-tree or similar)

# Sýn geymd á diski – Stored views

- Venjuleg sýn (view) er einfaldlega uppskrift af því hvernig mætti búa til vensl og fyrirspurnin er keyrð í hvert sinn sem við skoðum sýnina
- Stundum getur verið hagstætt að skrifa sýnina beint á disk, þá keyrir hún hraðar, ef við pössum að uppfæra sýnina í samræmi við uppfærslur á undirliggjandi gögnum
- Til dæmis gæti uppfærslan

```
INSERT INTO Movie
VALUES('Kill Bill',2003,123,1, 'Paramount',23456);
```

  valdið sjálfkrafa samsvarandi uppfærslu á sýninni ParamountMovie
- Það þarf því ekki að kosta mjög mikið að viðhalda gögnunum í sýninni

# Stored views

- A regular view is simply a recipe for creating a relation and the query is executed each time we use the view
- Sometimes it can be advantageous to write the view direcly to disk, then it executes faster, if we make sure to update the stored view in accordance with updates on the underlying data
- For example the update

```
INSERT INTO Movie
VALUES('Kill Bill',2003,123,1, 'Paramount',23456);
```

  could cause an automatic update on the view ParamountMovie
- It is not necessarily expensive to maintain the data in the stored view correctly

# Endurskrift fyrirspurna – Rewriting queries

- Ef fyrirspurn er hægvirk má spyrja gagnagrunninn hvað hann gerir til að framkvæma hana
  - Í SQLite getum við notað skipunina EXPLAIN QUERY PLAN SELECT ...
- Til að hraða algengum fyrirpurnum er oft nóg að
  - Bæta við lyklum og ytri lyklum
  - Bæta við vísum sem hjálpa til, sérstaklega fyrir tengingar (join)
- Þegar þetta dugar ekki er hægt að geyma niðurstöður (til dæmis tenging á fleiri en einni töflu) til seinni nota
- Þá þarf að geyma fyrirspurnina sem MATERIALIZED VIEW
  - SQLite gefur okkur ekki slíka virkni

# Rewriting queries

- If a query is slow we can ask the database what it does to execute it
  - In SQLite we can use the command EXPLAIN QUERY PLAN SELECT ...
- To speed up common queries it is often enough to
  - Add keys and foreign keys
  - Add indexes that help, especially for joins
- When this is not enough we can store results (for example joins of multiple tables) for later use
- Then we need to store the query as a MATERIALIZED VIEW
  - SQLite does not support this

# Uppskrift af endurskrift

- Ef við erum með fyrirspurn Q á sniðinu
  SELECT DQ FROM TQ WHERE CQ
- og materialized view, V, á sniðinu
  SELECT DV FROM TV WHERE CV
- þá getum við notað V inni í útreikningnum fyrir Q ef
  - allar töflur í TV koma fyrir í TQ
  - CQ er jafngilt CV AND C þar sem C eru einhver (kannski tóm) (viðbótar)skilyrði
  - ef C er ekki tómt þá eru dálkarnir sem koma fyrir í C líka í töflunum í TV
  - dálkarnir úr DQ sem eru úr töflum í TV koma líka fyrir í DV
- Með öðrum orðum: V má að vera víðtækara (breiðara og lengra) en einhver „undirfyrirspurn“ (ofanvarp) Q, og þá má fá Q sem útkomu með því að skera fyrst V niður, þ.e. með því að gera fyrirspurn í V, og gera síðan frekari tengingar til að fá jafngildi Q

# Recipe for rewriting

- If we have a query Q of the form
  SELECT DQ FROM TQ WHERE CQ
- And a materialized view, V, of the form
  SELECT DV FROM TV WHERE CV
- Then we can use V inside the computations for Q if
  - All tables in TV occur in TQ
  - CQ is is equivalent to CV AND C where C is some (perhaps empty) (additional)condition
  - If C is not empty then the columns that occur in C are also are in the tables of TV
  - The columns of DQ that are in the tables of TV also are in DV
- In other words: V may be more extensive (broader and longer) than some “subquery“ (projection) of Q, and then we can get Q as a result by first restricting V, i.e. By doing a query on V, and then do further joins to get the equivalent of Q

# Uppskrift af endurskrift

- Ef við erum með fyrirspurn Q á sniðinu
  SELECT DQ FROM TQ WHERE CQ
- og materialized view, V, á sniðinu
  SELECT DV FROM TV WHERE CV
- Ef öll skilyrðin eru uppfyllt og þar með að CQ sé jafngilt CV AND C þá má endurskrifa fyrirspurnina með því að
  - í stað TQ setja V og þær töflur í TQ sem ekki koma fyrir í TV
  - í stað CQ setja C, eða sleppa WHERE ef C er tómt

# Recipe for rewriting

- If we have a query Q of the form
  SELECT DQ FROM TQ WHERE CQ
- And a materialized view, V, of the form
  SELECT DV FROM TV WHERE CV
- If all conditions are fulfilled and thereby CQ is equivalent to CV AND C then we may rewrite the query by
  - Replacing TQ with V along with the tables in TQ that do not occur in TV
  - Replacing CQ with C, or omit WHERE if C is empty

# Dæmi

- Gerum ráð fyrir sýninni MovieProd
  CREATE MATERIALIZED VIEW MovieProd AS
  SELECT title,year,name FROM Movie,MovieExec
  WHERE cert=producerC;
- og fyrirspurninni
  SELECT starName FROM StarsIn,Movie,MovieExec
  WHERE movieTitle=title AND movieYear=year
  AND producerC=cert AND name='George Lucas';
- þá getum við endurskrifað hana sem
  SELECT starName FROM StarsIn,MovieProd
  WHERE movieTitle=title AND movieYear=year
  AND name='George Lucas';
- Þar með fækkum við tengingum úr þremur í tvær

# Example

- Assume the view MovieProd
  CREATE MATERIALIZED VIEW MovieProd AS
  SELECT title,year,name FROM Movie,MovieExec
  WHERE cert=producerC;
- And the query
  SELECT starName FROM StarsIn,Movie,MovieExec
  WHERE movieTitle=title AND movieYear=year
  AND producerC=cert AND name='George Lucas';
- Then we can rewrite it as
  SELECT starName FROM StarsIn,MovieProd
  WHERE movieTitle=title AND movieYear=year
  AND name='George Lucas';
- Thereby reducing the number of joins from three to two

# Ágæðun fyrirspurna
# Query Optimization

Hvernig framkvæmir gagnagrunnskerfið fyrirspurn og hvernig velur kerfið milli þeirra valkosta sem koma til greina til að framkvæma hana?

How does the database management system execute a query and how does it choose between the options available for executing it?

# Ágæðun fyrirspurna
# Query optimization

- Ágæðun fyrirspurnar – Query optimization
  - Að velja viðeigandi framkvæmdarás til að framkvæma fyrirspurn
    Choosing the appropriate sequence of events to execute a query
- Gagnagrunnskerfið vinnur úr fyrirspurn
  The DBMS executing a query
  1. Þýðir SQL fyrirspurnina í samsvarandi tré eða net með venslaalgebruaðgerðum
     Compiles the SQL query into a corresponding tree or graph with relational algebra operations
  2. Breytir trénu eða netinu í jafngilt tré eða net sem er hraðvirkara, ef unnt er
     Transforms the tree or graph into an equivalent tree or graph that is faster, if possible
     - að teknu tilliti til vísa, lykla , fjölda raða í töflum, röð gilda í skrám, o.s.frv.
       considering indexes, the count of rows in tables, the order of values in files, etc.
  3. Þýðir tréð eða netið í runu skipana sem gagnagrunnurinn getur framkvæmt
     Compiles the tree or graph into a sequence of operations that the DBMS can execute
  4. Keyrir fyrirspurnina með því að framkvæma skipanarununa
     Runs the query by executing the operation sequence

# Þýðing fyrirspurna í venslaalgebru

- Fyrirspurnarbálkur (query block) er grunneiningin sem SQL þýðandinn býr til venslaalgebrusegð fyrir
- Fyrirspurnarbálkur inniheldur staka SELECT-FROM-WHERE skipun, auk GROUP BY og HAVING ef þær klausur eru til staðar
- Faldaðar (nested) fyrirspurnir verða aðskildir fyrirspurnarbálkar
- Venslaalgebran þarf að vera útvíkkuð með hópunaraðgerðum til að styðja GROUP BY, SUM(), COUNT(), o.s.frv.

  Dæmi:

  1. $\gamma_{MAX(Salary)}(EMPLOYEE)$ skilar hæstu launum í EMPLOYEE töflunni (skilar reyndar töflu með einum dálki og einni röð)
  2. $\gamma_{Dno,SUM(Salary)}(EMPLOYEE)$ skilar töflu með heildarlaunum deilda (með tveimur dálkum, Dno og SUM_Salary)

# Compilation of queries in relational algebra

- A query block is the fundamental unit that the SQL compiler compiles a relational algebra expression for
- A query block contains a single SELECT-FROM-WHERE skipun, along with GROUP BY and HAVING klauses if they are used
- Nested queries become separate query blocks
- The relational algebra needs to be extended with aggregate operations to support GROUP BY, SUM(), COUNT(), etc.

  Example:

  1. $\gamma_{MAX(Salary)}(EMPLOYEE)$ returns the maximum salary in the EMPLOYEE table (actually returns a table with one column and one row)
  2. $\gamma_{Dno,SUM(Salary)}(EMPLOYEE)$ returns a table with the total salary of departments (with two columns, Dno and SUM_Salary)

```
SELECT Lname,Fname
FROM EMPLOYEE
WHERE Salary > (      SELECT MAX(Salary)
                      FROM EMPLOYEE
                      WHERE Dno=5);
```

```
SELECT Lname,Fname
FROM EMPLOYEE
WHERE Salary>C
```

$$\pi_{Lname,Fname}\left(\sigma_{Salary>C}(EMPLOYEE)\right)$$

```
SELECT MAX(Salary)
FROM EMPLOYEE
WHERE Dno=5
```

$$\gamma_{MAX(Salary)}\left(\sigma_{Dno=5}(EMPLOYEE)\right)$$

# Ytri röðun – External Sorting

- Ytri röðun (external sorting) er röðun á stórum skrám á diskum, sem ekki komast fyrir í innra minni tölvunnar (RAM)
  - Dæmigert vandamál í stórum gagnagrunnum
- Aðferð röðunar og samröðunar
  - Röðum fyrst litlum hlutum aðalskrárinnar inni í minni
  - Samröðum síðan litlum röðuðum bútum í stærri uns öll skráin er röðuð
  - Heildartímaflækjan er $O(n \log n)$
  - Hagstæðara að raða sem mest í innra minni (RAM)

# External Sorting

- External sorting is the sorting on disks (or other external storage), of large files that do not fit in RAM
  - Typical task in large databases
- Uses sorting and merging
  - First sort small segments of the file in RAM
  - Merge the small segments into larger segments until fully sorted
  - Total time complexity is $O(n \log n)$
  - Advantageous to sort as much as possible inside RAM

# Valkostir fyrir val- og tengiaðgerðir
# Options for select and join operations

- Íhugum fyrst valaðgerðir (SELECT eða $\sigma$) – Consider $\sigma$
- Dæmi/Example
  1. $\sigma_{Ssn=123456789}(EMPLOYEE)$
  2. $\sigma_{Dnumber>5}(DEPARTMENT)$
  3. $\sigma_{Dno=5}(EMPLOYEE)$
  4. $\sigma_{Dno=5 \wedge Salary>30000 \wedge Sex='F'}(EMPLOYEE)$
  5. $\sigma_{Essn=123456789 \wedge Pno=10}(WORKS\_ON)$

# Helstu valkostir fyrir valaðgerð
# Options for select operation

1. Línuleg leit – Linear search
2. Helmingunarleit – Binary search
3. Nota einkvæman vísi, til dæmis B-tré eða tætitöflu, til að sækja stakar færslur – Use a unique index, e.g. a B-tree or hashtable, to fetch individual records
4. Nota einkvæman vísi (til dæmis B-tré) til að sækja margar færslur – Use a unique index (e.g. a B-tree) to fetch multiple records
5. Nota hópavísi til að finna margar færslur – Use a group index to find many records
6. Nota samsettan vísi til að leita samkvæmt skilyrði á fleiri en einn dálk – Use a combined index to search according to a condition on more than one column
7. Fyrir flókin skilyrði þarf oft að gera runu valaðgerða – For complex conditions we often need a sequence of selection operations

# Val á valkostum fyrir valaðgerð
# Choosing the option for a selection operation

- Þegar valaðgerðin hefur einstakt skilyrði athugum við hvort til er aðgangsleið samkvæmt skilyrðinu (til dæmis vísir eða hvort undirliggjandi skrá er í röð samkvæmt skilyrðinu)
  When the selection operation has a single condition we check whether there is an access path according to the condition (for example an index or the underlying file is ordered according to the condition)
- Ef slík aðgangsleið er til staðar þá notum við hana, annars verðum við að nota línulega leit
  If such an access path exists we use it, otherwise we fall back to linear search
- Ef skilyrðið er samsett með **og** rökaðgerð þá er hagstætt að nota fyrst þann hluta skilyrðisins sem gefur sem fæstar niðurstöður á sem stystum tíma
  If the condition is composite with an **and** operation then it is good to use first the part of the condition that results in the fewest results in the shortest time
- Samsett skilyrði með **eða** rökaðgerð þurfa aðra meðhöndlun
  A composite condition with **or** operation needs other handling

# Tengiaðgerðir – JOIN operations

- Einfaldast er jafntenging tveggja taflna (equijoin)
  Equijoin is simplest
- En tengingar geta haft fleiri en tvær töflur og flókin skilyrði
  But joins can have more than one table and complicated conditions

- Dæmi um einfaldar tengingar – Examples of simple joins

1. $EMPLOYEE \bowtie_{Dno=Dnumber} DEPARTMENT$
2. $DEPARTMENT \bowtie_{MgrSsn=Ssn} EMPLOYEE$

# Útfærsluvalkostir fyrir tengiaðgerðir $R \bowtie_\theta S$

1. Tenging með földuðum lykkjum (brute force)
   - Lykkjum í gegnum sérhvert gildi í *R* og í sérhverri umferð í þeirri lykkju, lykkjum í gegnum sérhvert gildi í *S* og athugum hvort skilyrðið er uppfyllt
2. Tenging með einfaldri lykkju
   - Lykkjum gegnum sérhvert gildi í annarri töflunni og fyrir sérhvert slíkt gildi notum við tiltæka aðgangsleið (B-tré, tætitafla, ...) í hina töfluna
   - Ef engin slík aðgangsleið er tiltæk getur verið hagstætt að búa hana til tímabundið (helst fyrir smærri töfluna)
3. Samröðunartenging
   - Ef færslurnar í *R* og *S* eru í röð sem samsvarar $\theta$ þá getum við lesið skrárnar samtímis í vaxandi eða minnkandi röð
   - Ef ekki þá getum við stundum raðað *R* og *S* svo þetta sé hægt
   - Aðeins þarf þá að lesa einu sinni gegnum hvora skrá (töflu) til að tengja

# Implementation choices for join $R \bowtie_\theta S$

1. Joining using nested loops (brute force)
   - Loop through each row in *R* and in each pass through that loop, loop through each row in *S* and check whether the condition is fulfilled
2. Joining with a single loop
   - Loop through each row in one of the tables and for each row use an available access path (B-tree, hashtable, …) into the other table
   - If no such access path is available it may be advantageous to create it temporarily (preferably for the smaller table)
3. Joining by merging
   - If the rows in *R* and *S* are ordered in an order that corresponds to $\theta$ then we can read the tables in parallel in ascending or descending order
   - If not, we can sometimes sort *R* and *S* to make this possible
   - Only need to read once through each table to join

# Ofanvarp $-\ \pi_{<eiginleikalisti>}(R)$

- Ef <*eiginleikalisti*> innifelur lykil venslanna R þá afritum við einfaldlega allar n-dir úr R í niðurstöðuna, en sleppum þeim eiginleikum sem ekki eru í listanum
- Ef <*eiginleikalisti*> innifelur ekki lykil R þá þarf e.t.v. að eyða endurteknum n-dum
- Aðferðir til að eyða endurtekningum:
  - Röðun
  - Tæting

# Projection $-$ $\pi_{<attribute\ list>}(R)$

- If <*attribute list*> contains the key or the relation then we simply copy all tuples from $R$ into the result, but skip the attributes that are not in the list
- If <*attribute list*> does not contain a key for $R$ then we may need to delete repeated tuples
- Methods for deleting repeats:
  - Sorting
  - Hashing

# Mengjaaðgerðir – Set Operations

- Sammengi, sniðmengi, mengjamismunur – Unions, intersections, set difference
  - Þessar aðgerðir má allar útfæra með því að raða báðum töflunum og síðan framkvæma nokkurs konar samröðun (merge)
    Sort both tables and then do a variant of merge
  - Útkoman inniheldur ýmist öll gildi, sameiginleg gildi eða gildi úr öðru sem ekki eru í hinu
    The result contains either all values, common values, or values from one not in the other
- Mengjamargfeldi – Cross product
  - Ef $R$ inniheldur $n$ raðir og $j$ eiginleika, $S$ inniheldur $m$ raðir og $k$ eiginleika þá inniheldur $R \times S$ $nm$ raðir og $j + k$ eiginleika
    if $R$ contains $n$ rows and $j$ attributes, $S$ contains $m$ rows and $k$ attributes then $R \times S$ contains $nm$ rows and $j + k$ attributes
  - Mengjamargfeldið er því mjög dýrt og við ættum að forðast það í lengstu lög
    The cross product is therefore very expensive and we should avoid it if possible

# Hópaðgerðir – Group Operations

- Hópaðgerðir
  - MIN, MAX, SUM, COUNT, AVG
- Útfærsluaðferðir
  - Lestur gegnum skrár (töflur)
  - Vísar
- Dæmi

  SELECT MAX(Salary) FROM EMPLOYEE
- Ef til er vísir (í vaxandi röð) á Salary fyrir EMPLOYEE venslin þá má finna hágildið á einfaldan hátt, annars þarf kannski að lesa gegnum alla töfluna
- Fyrir aðrar hópaðgerðir svo sem SUM, COUNT og AVG er stundum hægt að nota vísa til að hraða vinnslunni
- GROUP BY flækir málið, en samt má stundum nota vísa í útfærslu

# Group Operations

- Aggregate operations
  - MIN, MAX, SUM, COUNT, AVG
- Implementation methods
  - Traverse tables whole tables
  - Use indexes
- Example

  SELECT MAX(Salary) FROM EMPLOYEE
- If there is an index (in ascending order) on Salary for the EMPLOYEE relation then we can find the maximum in a simple way, otherwise we may need to read through the whole table
- For other aggregate operations such as SUM, COUNT and AVG we can sometimes speed up the processing using indexes
- GROUP BY complicates matters, but we still may be able to utilize indexes

# Útfærsla ytri tenginga – Outer Joins

- Ytri tengingar
  - LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN
- Full ytri tenging er sammengi hinna ytri tenginganna
- Dæmi:

  ```
  SELECT Fname, Dname
  FROM (EMPLOYEE LEFT OUTER JOIN DEPARTMENT ON Dno=Dnumber);
  ```

- Tryggt er að niðurstaðan innihaldi allar n-dir úr EMPLOYEE
  - Ef viðkomandi n-d hefur ekki samsvarandi n-d í DEPARTMENT þá eru sett NULL gildi í útkomuna í dálkana sem kæmu frá DEPARTMENT
- Útfærsla ytri tenginga er svipuð og fyrir venjulegar tengingar
  - Aðlaga þarf faldaðar lykkjur og tengingar með röðun og samröðun í samræmi við eilítið breyttar kröfur

# Implementing Outer Joins

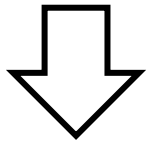
- Outer joins
  - LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN
- The full outer join is the set union of the other outer joins
- Example:

  ```
  SELECT Fname, Dname
  FROM (EMPLOYEE LEFT OUTER JOIN DEPARTMENT ON Dno=Dnumber);
  ```

- Ensures the result contains all tuples from EMPLOYEE
  - If the tuple does not have a corresponding tuple in DEPARTMENT then NULL values are put into the columns that would have come from DEPARTMENT
- The implementation is similar as for other joins
  - Nested loops need adjustments and joins with sorting and merging need slight adjustments

# Dæmi um útfærslu ytri tengingar með einfaldri venslaalgebru (án ⟗)

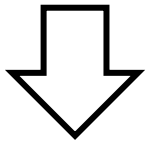
```
SELECT Fname, Dname
FROM (EMPLOYEE LEFT OUTER JOIN DEPARTMENT ON Dno=Dnumber);
```



$$TEMP1 \leftarrow \pi_{Fname,Dname}(EMPLOYEE \bowtie_{Dno=Dnumber} DEPARTMENT)$$
$$TEMP2 \leftarrow \pi_{Fname}(EMPLOYEE) - \pi_{Fname}(TEMP1)$$
$$TEMP3 \leftarrow TEMP2 \times \{(\bot)\}$$
$$RESULT \leftarrow TEMP1 \cup TEMP3$$

# Example of implementing an outer join with simple relational algebra (without ⟕)

```
SELECT Fname, Dname
FROM (EMPLOYEE LEFT OUTER JOIN DEPARTMENT ON Dno=Dnumber);
```



$$TEMP1 \leftarrow \pi_{Fname,Dname}(EMPLOYEE \bowtie_{Dno=Dnumber} DEPARTMENT)$$
$$TEMP2 \leftarrow \pi_{Fname}(EMPLOYEE) - \pi_{Fname}(TEMP1)$$
$$TEMP3 \leftarrow TEMP2 \times \{(\perp)\}$$
$$RESULT \leftarrow TEMP1 \cup TEMP3$$

# Samsetning aðgerða með keðjun

- Vandamál
  - Fyrirspurnir verða að runum (venslaalgebru)aðgerða
  - Hver aðgerð gefur af sér milliniðurstöðu
  - Geymsla milliniðurstaðna á diski er tímafrekt og dýrt
- Möguleg lausn
  - Forðumst að geyma milliniðurstöður
  - Keyrum þess í stað gögnin viðstöðulaust gegnum keðju aðgerða

# Chaining operations

- Challenge
  - Queries become a sequence of relational algebra operations
  - Each operation produces an intermediate (temporary) result
  - Storing temporary results is time consuming and expensive
- Possible solution
  - Avoid storing intermediate results
  - Instead pass the data uninterrupted through a chain of operations

# Samsetning aðgerða með keðjun

- Dæmi
  - Í tengingu tveggja vensla má keðja saman tvær valaðgerðir og eina eða fleiri ofanvarpsaðgerð ásamt tengiaðgerðinni
- Aðgerðir má þýða í gagnagrunnskerfinu á þann hátt að þær séu keðjaðar saman
- Niðurstöður valaðgerða eru sendar í keðju til tengiaðgerðar
- Stundum er þetta kallað straumavinnsla (*stream-based processing* – sbr. strauma í forritunarmálum)

# Chaining operations

- Example
  - When joining two relations we can chain together two selection operations and one or more projection operations along with the join operation
- The operations can be compiled by the DBMS so that they are chained
- Results from selections are passed through the chain to a join operaiton
- Sometimes this is called *stream-based processing* – similar to streams in programming languages

# Þumalputtareglur í ágæðun fyrirspurna Heuristics

- Ágæðunarferli
  - Þýðandinn (hluti gagnagrunnskerfisins) smíðar milliþulu (*intermediate code*, oftast einhvers konar tré), sem stendur fyrir fyrirspurnina
  - Beitum þumalputtareglum (*heuristics*) til að ágæða milliþuluna
  - Framleiðum svo fyrirspurnaráætlun (query execution plan) sem byggt er á þeim aðgangsleiðum sem tiltækar eru á þær skrár og töflur sem notaðar eru í fyrirspurninni
- Helsta þumalputtareglan er að beita fyrst þeim aðgerðum sem minnka milliniðurstöðurnar (minnka magn gagna)
  - Til dæmis viljum við beita val- og ofanvarpsaðgerðum áður en við beitum tengiaðgerðum og öðrum tvíundaraðgerðum

# Heuristics for optimizing

- Optimization process
  - The compiler (a part of the DBMS) produces *intermediate code*, (usually some sort of tree), that represents the query
  - Use *heuristics* to optimize the intermediate code
  - Then produce a query execution plan based on the access paths that are available for the files and tables used in the query
- The primary heuristic is to first apply the operations that reduce the amount of data
  - For example we want to apply selection and projection operations before we apply join operations and other binary operations

# Kostnaðaráætlanir fyrirspurna
# Cost Estimates

- Fyrir tiltekna SQL fyrirspurn eru almennt margar mögulegar runur venslaalgebruaðgerða sem geta reiknað fyrirspurnina
- Gagnagrunnskerfið athugar eitthvað hlutmengi af öllum möguleikum
- Gagnagrunnskerfið metur kostnað hvers möguleika
  - Á grundvelli þeirra aðgerða sem framkvæmdar eru, tiltækra aðgangsleiða og eiginleika þeirra, vitneskju um stærð skráa og taflna og e.t.v. fleiri atriða
- Gagnagrunnskerfið velur þann möguleika sem hagstæðastur er, miðað við gefið kostnaðarmat

# Cost Estimates for queries

- For a given query there are in general many possible sequences of relational algebra operations that can compute the query
- The DBMS will consider some subset of all possibilities
- Te DBMS will estimate the cost of each option
  - On the basis of the operations performed, available access paths and their properties, knowledge about the sizes of files and relations and perhaps other issues
- The DBMS will select the choice that is most advantageous given the cost estimate

# Helstu kostnaðarþættir
# Common Cost Factors

- Kostnaðarþættir – Cost Factors
  - Aðgangur í ytri geymslur – Access to external storage (diskahraði – disk speed)
  - Diskapláss (disk space)
  - Reiknikostnaður (CPU cost)
  - Minniskostnaður (RAM cost)
  - Samskiptakostnaður (network cost)
- Mismunandi gagnagrunnskerfi leggja mismunandi áherslur á mismunandi kostnaðarþætti
  Different DBMS‘s emphasize different cost factors differently