CS 564 Midterm Review

The Best Of Collection (Master Tracks), Vol. 1

Announcements

- Midterm next Wednesday
- In class: roughly 70 minutes
 - Come in 10 minutes earlier!

Midterm

- Format:
 - Regular questions. No multiple-choice.
 - This implies fewer questions $\textcircled{\sc op}$
 - A couple bonus questions
 - Closed book and no aids! We will provide a cheat sheet
- Material: Everything including buffer management
 - External sort is not fair game!
- High-lights:
 - Simple SQL and Schema Definitions
 - Join Semantics
 - Function Dependencies and Closures
 - Decompositions (BCNF and Properties)
 - Buffer Pool and Replacement Policies

High-Level: SQL

- Basic terminology:
 - relation / table (+ "instance of"), row / tuple, column / attribute, multiset
- Table schemas in SQL
- Single-table queries:
 - SFW (selection + projection)
 - Basic SQL operators: LIKE, DISTINCT, ORDER BY
- Multi-table queries:
 - Foreign keys
 - JOINS:
 - Basic SQL syntax & semantics of

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	\$19.99	GizmoWorks
Powergizmo	\$29.99	GizmoWorks
SingleTouch	\$149.99	Canon
MultiTouch	\$203.99	Hitachi

A <u>relation</u> or <u>table</u> is a multiset of tuples having the attributes specified by the schema

A <u>multiset</u> is an unordered list (or: a set with multiple duplicate instances allowed)

A <u>tuple</u> or <u>row</u> is a single entry in the table having the attributes specified by the schema

An <u>attribute</u> (or <u>column</u>) is a typed data entry present in each tuple in the relation

Table Schemas

• The **schema** of a table is the table name, its attributes, and their types:

Product(Pname: string, Price: float, Category: string, Manufacturer: string)

• A key is an attribute whose values are unique; we underline a key

Product(<u>Pname</u>: string, Price: float, Category: string, <u>Manufacturer</u>: string)

SQL Query

• Basic form (there are many many more bells and whistles)

SELECT <attributes>
FROM <one or more relations>
WHERE <conditions>

Call this a <u>SFW</u> query.

LIKE: Simple String Pattern Matching

SELECT	*	
FROM	Products	
WHERE	PName LIKE	'%gizmo%'

DISTINCT: Eliminating Duplicates

SELECT DISTINCT CategoryFROMProduct

ORDER BY: Sorting the Results

SELECT	PName, Price
FROM	Product
WHERE	Category='gizmo'
ORDER BY	Price, PName

Joins

Product

PName	Price	Category	Manuf	Compar			Company
Gizmo	\$10	Gadgets	GWorks		Cname	Stock	Country
		Gaugets	GIVE 1		GWorks	25	USA
Powergizmo	\$29	Gadgets	Gadgets GWorks		Canon	65	lanan
SingleTouch	\$149	Photography	Canon			1.5	Japan
MultiTouch	\$203	Household	Hitachi		Hitachi	15	lapan
ivitaliti i Odelli	\$203	mousemond	Indem				

SELECT PName, Price
FROM Product, Company
WHERE Manufacturer = CName
AND Country='Japan'
AND Price <= 200</pre>

PName	Price
SingleTouch	\$149.99

An example of SQL semantics



10

High-Level: Advanced SQL

- Set operators
 - INTERSECT, UNION, EXCEPT, [ALL]
 - Subtleties of multiset operations
- Nested queries
 - IN, ANY, ALL, EXISTS
 - Correlated queries
- Aggregation
 - AVG, SUM, COUNT, MIN, MAX, ...
- GROUP BY
- NULLs & Outer Joins

An Unintuitive Query





UNION

EXCEPT

SELECTR.AFROMR, SWHERER.A=S.AINTERSECTSELECTR.AFROMR, TWHERER.A=T.A

INTERSECT



SELECT R.A FROM R, S WHERE R.A=S.A UNION SELECT R.A FROM R, T WHERE R.A=T.A



SELECTR.AFROMR, SWHERER.A=S.AEXCEPTSELECTR.AFROMR, TWHERER.A=T.A



Nested queries: Sub-queries Returning Relations

Company(<u>name</u>, city)
Product(<u>name</u>, maker)
Purchase(<u>id</u>, product, buyer)

```
SELECT c.city
FROM Company c
WHERE c.name IN (
    SELECT pr.maker
    FROM Purchase p, Product pr
    WHERE p.product = pr.name
    AND p.buyer = 'Joe Blow')
```

"Cities where one can find companies that manufacture products bought by Joe Blow"

Nested Queries: Operator Semantics

Product(name, price, category, maker)

ALL

SELECT	name
FROM	Product
WHERE	price > ALL(
SELEC	T price
FROM	Product
WHERE	maker = 'G')

ANY

EXISTS

SELECT	name
FROM	Product
WHERE	price > ANY(
SELEC	T price
FROM	Product
WHERE	maker = 'G')

SELECT name
FROM Product p1
WHERE EXISTS (
 SELECT *
 FROM Product p2
 WHERE p2.maker = 'G'
 AND p1.price =
 p2.price)

Find products that are more expensive than *all products* produced by "G" Find products that are more expensive than *any one product* produced by "G" Find products where *there exists some* product with the same price produced by "G"

Nested Queries: Operator Semantics

Product(name, price, category, maker)

ALL

SELECT name
FROM Product
WHERE price > ALL(X)

ANY

SELECT name
FROM Product
WHERE price > ANY(X)

EX	IS7	٢S
ΕX	12	15

SELECTnameFROMProduct p1WHEREEXISTS (X)

Price must be > *all* entries in multiset X Price must be > *at least one* entry in multiset X X must be non-empty

*Note that p1 can be referenced in X (correlated query!)



Movie(title, year, director, length)



Find movies whose title appears more than once.

Note the scoping of the variables!

Note also: this can still be expressed as single SFW query...

Simple Aggregations

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

SELECT	SUM(price	*	quantity)
FROM	Purchase		
WHERE	<pre>product =</pre>	'k	bagel'

Grouping & Aggregations: GROUP BY

SELECT	<pre>product, SUM(price*quantity)</pre>
FROM	Purchase
WHERE	date > '10/1/2005'
GROUP BY	product
HAVING	SUM(quantity) > 10

Find total sales after 10/1/2005, only for products that have more than 10 total units sold

HAVING clauses contains conditions on aggregates

Whereas WHERE clauses condition on *individual tuples...*

GROUP BY: (1) Compute FROM-WHERE

SELECT	<pre>product, SUM(price*quantity) AS TotalSales</pre>
FROM	Purchase
WHERE	date > '10/1/2005'
GROUP BY	product
HAVING SU	JM(quantity) > 10

FROM WHERE

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10
Craisins	11/1	2	5
Craisins	11/3	2.5	3

GROUP BY: (2) Aggregate by the GROUP BY

SELECT FROM	<pre>product, SUM(price*quantity) AS TotalSales Purchase</pre>
WHERE	date > '10/1/2005'
GROUP BY HAVING SU	product M(quantity) > 10

Product	Date	Price	Quantity	
Bagel	10/21	1	20	GROUP BY
Bagel	10/25	1.50	20	
Banana	10/3	0.5	10	
Banana	10/10	1	10	
Craisins	11/1	2	5	
Craisins	11/3	2.5	3	

Product	Date	Price	Quantity
	10/21	1	20
Bager	10/25	1.50	20
D	10/3	0.5	10
Banana	10/10	1	10
Craisins	11/1	2	5
	11/3	2.5	3

GROUP BY: (3) Filter by the HAVING clause

HAVING

SELECT	<pre>product, SUM(price*quantity) AS TotalSales</pre>
FROM	Purchase
WHERE	date > '10/1/2005'
GROUP BY	product
HAVING SU	JM(quantity) > 30

Product	Date	Price	Quantity
D 1	10/21	1	20
Bager	10/25	1.50	20
D	10/3	0.5	10
Banana	10/10	1	10
Croising	11/1	2	5
Craisins	11/3	2.5	3

	Product	Date	Price	Quantity
	Bagel	10/21	1	20
		10/25	1.50	20
	Banana	10/3	0.5	10
		10/10	1	10

GROUP BY: (3) SELECT clause

SELECT	<pre>product, SUM(price*quantity) AS TotalSales</pre>
FROM	Purchase
WHERE	date > '10/1/2005'
GROUP BY	product
HAVING SU	JM(quantity) > 100

Product	Date	Price	Quantity
Dagal	10/21	1	20
Bager	10/25	1.50	20
Danana	10/3	0.5	10
Banana	10/10	1	10

SELECT	Product	TotalSales
	Bagel	50
V	Banana	15

General form of Grouping and Aggregation



Evaluation steps:

- 1. Evaluate FROM-WHERE: apply condition C_1 on the attributes in $R_1, ..., R_n$
- 2. **GROUP BY** the attributes a_1, \dots, a_k
- 3. Apply HAVING condition C_2 to each group (may have aggregates)
- 4. Compute aggregates in SELECT, S, and return the result

Null Values

- For numerical operations, NULL -> NULL:
 - If x = NULL then $4^{*}(3-x)/7$ is still NULL
- For boolean operations, in SQL there are three values:

FALSE	=	0
UNKNOWN	=	0.5
TRUE	=	1

• If x= NULL then x="Joe" is UNKNOWN

Null Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 C1

```
SELECT *
FROM Person
WHERE (age < 25)
AND (height > 6 AND weight > 190)
```

Won't return e.g. (age=20 height=NULL weight=200)!

Rule in SQL: include only tuples that yield TRUE / 1.0

Null Values

Unexpected behavior:





Some Persons are not included !

Now it includes all Persons!

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

RECAP: Inner Joins

By default, joins in SQL are "inner joins":

Product(name, category)
Purchase(prodName, store)

SELECT Product.name, Purchase.store

FROM Product

JOIN Purchase **ON** Product_name = Purchase_prodName

SELECT Product.name, Purchase.store

FROM Product, Purchase

WHERE Product_name = Purchase_prodName

Both equivalent: Both INNER JOINS!

INNER JOIN:

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

SELECT	Product.name, Purchase.store
FROM	Product
INNE	R JOIN Purchase
	<pre>ON Product.name = Purchase.prodName</pre>

Note: another equivalent way to write an INNER JOIN!

name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

LEFT OUTER JOIN:

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

General clarification: Sets vs. Multisets

- In theory, and in any more formal material, <u>by definition</u> all relations are <u>sets</u> of tuples
- In SQL, relations (i.e. tables) are multisets, meaning you can have duplicate tuples
 - We need this because intermediate results in SQL don't eliminate duplicates
- If you get confused: just state your assumptions & we'll be forgiving!

High-Level: ER Diagrams

- ER diagrams!
 - Entities (vs. Entity Sets)
 - Relationships
 - Multiplicity
 - Constraints: Keys, single-value, referential, participation, etc...





What is a Relationship?



A <u>relationship</u> between entity sets P and C is a *subset of all possible pairs of entities in P and C,* with tuples uniquely identified by *P and C's keys*

What is a Relationship?





A <u>relationship</u> between entity sets P and C is a *subset of all possible pairs of entities in P and C,* with tuples uniquely identified by *P and C's keys*

Company C \times Product P

<u>C.name</u>	P.name	P.category	P.price
GizmoWorks	Gizmo	Electronics	\$9.99
GizmoWorks	GizmoLite	Electronics	\$7.50
GizmoWorks	Gadget	Toys	\$5.50
GadgetCorp	Gizmo	Electronics	\$9.99
GadgetCorp	GizmoLite	Electronics	\$7.50
GadgetCorp	Gadget	Toys	\$5.50

Makes

<u>C.name</u>	P.name
GizmoWorks	Gizmo
GizmoWorks	GizmoLite
GadgetCorp	Gadget

Multiplicity of E/R Relationships

One-to-one:

Many-to-one:

One-to-many:

Many-to-many:



Indicated using arrows

X -> Y means <u>there exists a</u> <u>function mapping</u> <u>from X to Y (recall</u> the definition of a function)
Constraints in E/R Diagrams

- Finding constraints is part of the E/R modeling process. Commonly used constraints are:
 - <u>Keys</u>: Implicit constraints on uniqueness of entities
 - Ex: An SSN uniquely identifies a person
 - <u>Single-value constraints:</u>
 - Ex: a person can have only one father
 - <u>Referential integrity constraints:</u> Referenced entities must exist
 - Ex: if you work for a company, it must exist in the database
 - <u>Other constraints:</u>
 - Ex: peoples' ages are between 0 and 150

Recall

KEYs!

FOREIGN

RECALL: Mathematical def. of Relationship

- A mathematical definition:
 - Let A, B be sets
 - A={1,2,3}, B={a,b,c,d},
 - A x B (the *cross-product*) is the set of all pairs (a,b)
 - $A \times B = \{(1,a), (1,b), (1,c), (1,d), (2,a), (2,b), (2,c), (2,d), (3,a), (3,b), (3,c), (3,d)\}$
 - We define a <u>relationship</u> to be a subset of A x B
 - $R = \{(1,a), (2,c), (2,d), (3,b)\}$



RECALL: Mathematical def. of Relationship

• A mathematical definition:

- Let A, B be sets
- A x B (the *cross-product*) is the set of all pairs
- A <u>relationship</u> is a subset of A x B
- Makes is relationship- it is a *subset* of Product × Company:





RECALL: Mathematical def. of Relationship

- There can only be **one relationship for every unique combination of entities**
- This also means that the relationship is uniquely determined by the keys of its entities
- Example: the key for Makes (to right) is {Product.name, Company.name}



Why does this make sense?

This follows from our mathematical definition of a relationship- it's a SET!

High-Level: DB Design

- Redundancy & data anomalies
- Functional dependencies
 - For database schema design
 - Given set of FDs, find others implied- using Armstrong's rules
- Closures
 - Basic algorithm
 - To find all FDs
- Keys & Superkeys

Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes *anomalies*:

Similarly, we can't reserve a room without students = an <u>insert</u> anomaly

... CS229 C12

Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
		••

If every course is in only one room, contains <u>redundant</u> information!

If we update the room number for one tuple, we get inconsistent data = an <u>update anomaly</u>

If everyone drops the class, we lose what room the class is in! = a <u>delete anomaly</u>

Constraints Prevent (some) Anomalies in the Data

Student	Course
Mary	CS145
Joe	CS145
Sam	CS145
••	••

Course	Room
CS145	B01
CS229	C12

Is this form better?

- Redundancy?
- Update anomaly?
- Delete anomaly?
- Insert anomaly?

A Picture Of FDs



<u>Defn (again):</u> Given attribute sets $A=\{A_1,...,A_m\}$ and $B = \{B_1,...,B_n\}$ in R,

The *functional dependency* $A \rightarrow B$ on **R** holds if for *any* t_i, t_j in R:

 $\underline{if} t_i[A_1] = t_j[A_1] \text{ AND } t_i[A_2] = t_j[A_2] \text{ AND}$ $... \text{ AND } t_i[A_m] = t_j[A_m]$

 $\underline{\text{then}} t_i[B_1] = t_j[B_1] \text{ AND } t_i[B_2] = t_j[B_2]$ AND ... AND $t_i[B_n] = t_j[B_n]$

FDs for Relational Schema Design

- High-level idea: why do we care about FDs?
 - 1. Start with some relational *schema*
 - 2. Find out its *functional dependencies (FDs)*
 - 3. Use these to design a better schema
 - 1. One which minimizes possibility of anomalies

This part can be tricky!

Finding Functional Dependencies

Equivalent to asking: Given a set of FDs, $F = {f_1, ..., f_n}$, does an FD g hold?

Inference problem: How do we decide?

Answer: Three simple rules called **Armstrong's Rules.**

- 1. Split/Combine,
- 2. Reduction, and
- **3.** Transitivity... ideas by picture

Closure of a set of Attributes

Given a set of attributes $A_1, ..., A_n$ and a set of FDs F: Then the <u>closure</u>, $\{A_1, ..., A_n\}^+$ is the set of attributes B s.t. $\{A_1, ..., A_n\} \rightarrow B$

<u>Example:</u>	F =	<pre>{name} → {color} {category} → {department} {color, category} → {price}</pre>
Example		<pre>{name} = {name, color}</pre>

Closures:

{name}* = {name, color}
{name, category}* =
{name, category, color, dept, price}
{color}* = {color}

Closure Algorithm

Start with $X = \{A_1, ..., A_n\}$, FDs F. **Repeat until** X doesn't change; **do**: **if** $\{B_1, ..., B_n\} \rightarrow C$ is in F **and** $\{B_1, ..., B_n\} \subseteq X$: **then** add C to X. **Return** X as X⁺

 $\{name\} \rightarrow \{color\}$

{category} \rightarrow {dept}

{color, category} →
{price}

{name, category}* =
{name, category}

{name, category}* =
{name, category, color}

{name, category}⁺ =
{name, category, color, dept}

{name, category}* =
{name, category, color, dept,
price}

Keys and Superkeys

A <u>superkey</u> is a set of attributes $A_1, ..., A_n$ s.t. for *any other* attribute **B** in R, we have $\{A_1, ..., A_n\} \rightarrow B$

I.e. all attributes are *functionally determined* by a superkey

A **<u>key</u>** is a *minimal* superkey

Meaning that no subset of a key is also a superkey

CALCULATING Keys and Superkeys

• Superkey?

- Compute the closure of A
- See if it = the full set of attributes

• <u>Key?</u>

- Confirm that A is superkey
- Make sure that no subset of A is a superkey
 - Only need to check one 'level' down!

Let A be a set of attributes, R set of all attributes, F set of FDs:

IsSuperkey(A, R, F): A⁺ = *ComputeClosure*(A, F) Return (A⁺==R)?

IsKey(A, R, F): If not *IsSuperkey*(A, R, F): return False For B in *SubsetsOf*(A, size=len(A)-1): if IsSuperkey(B, R, F): return False return True

Also see Lecture-5.ipynb!!!

High-Level: Decompositions

- Conceptual design
- Boyce-Codd Normal Form (BCNF)
 - Definition
 - Algorithm
- Decompositions
 - Lossless vs. Lossy
 - A problem with BCNF

Back to Conceptual Design

Now that we know how to find FDs, it's a straight-forward process:

- 1. Search for "bad" FDs
- 2. If there are any, then *keep decomposing the table into sub-tables* until no more bad FDs
- 3. When done, the database schema is *normalized*

Recall: there are several normal forms...

Boyce-Codd Normal Form

BCNF is a simple condition for removing anomalies from relations:

A relation R is <u>in BCNF</u> if: if $\{A_1, ..., A_n\} \rightarrow B$ is a *non-trivial* FD in R then $\{A_1, ..., A_n\}$ is a superkey for R

Equivalently: \forall sets of attributes X, either (X⁺ = X) or (X⁺ = all attributes)

In other words: there are no "bad" FDs

Example

Name	SSN	PhoneNumber	City
Fred	123-45-6789	206-555-1234	Seattle
Fred	123-45-6789	206-555-6543	Seattle
Joe	987-65-4321	908-555-2121	Westfield
Joe	987-65-4321	908-555-1234	Westfield

 $\{SSN\} \rightarrow \{Name, City\}$

This FD is *bad* because it is <u>not</u> a superkey

 \Rightarrow <u>Not</u> in BCNF

What is the key? {SSN, PhoneNumber}

Example

Name	<u>SSN</u>	City
Fred	123-45-6789	Seattle
Joe	987-65-4321	Madison

<u>SSN</u>	PhoneNumber
123-45-6789	206-555-1234
123-45-6789	206-555-6543
987-65-4321	908-555-2121
987-65-4321	908-555-1234

{SSN} → {Name,City}

This FD is now good because it is the key

Let's check anomalies:

- Redundancy ?
- Update ?
- Delete ?

BCNFDecomp(R):

```
BCNFDecomp(R):
Find a set of attributes X s.t.: X<sup>+</sup> ≠ X and X<sup>+</sup> ≠
[all attributes]
```

Find a set of attributes X which has non-trivial "bad" FDs, i.e. is not a superkey, using closures

```
BCNFDecomp(R):
Find a set of attributes X s.t.: X<sup>+</sup> ≠ X and X<sup>+</sup> ≠
[all attributes]
```

```
if (not found) then Return R
```

If no "bad" FDs found, in BCNF!

```
BCNFDecomp(R):
Find a set of attributes X s.t.: X<sup>+</sup> ≠ X and X<sup>+</sup> ≠
[all attributes]
```

if (not found) then Return R

<u>let</u> $Y = X^+ - X$, $Z = (X^+)^C$

Let Y be the attributes that *X* functionally determines (+ that are not in X)

And let Z be the other attributes that it doesn't

```
BCNFDecomp(R):
Find a set of attributes X s.t.: X<sup>+</sup> ≠ X and X<sup>+</sup> ≠
[all attributes]
```

if (not found) then Return R

<u>let</u> $Y = X^+ - X$, $Z = (X^+)^C$ decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$ Split into one relation (table) with X plus the attributes that X determines (Y)...



```
BCNFDecomp(R):
Find a set of attributes X s.t.: X<sup>+</sup> ≠ X and X<sup>+</sup> ≠
[all attributes]
```

if (not found) then Return R

<u>let</u> $Y = X^+ - X$, $Z = (X^+)^C$ decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$ And one relation with X plus the attributes it *does not* determine (Z)



```
BCNFDecomp(R):
Find a set of attributes X s.t.: X<sup>+</sup> ≠ X and X<sup>+</sup> ≠
[all attributes]
```

if (not found) then Return R

<u>let</u> $Y = X^+ - X$, $Z = (X^+)^C$ decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

Return BCNFDecomp(R₁), BCNFDecomp(R₂)

Proceed recursively until no more "bad" FDs!

Example

```
BCNFDecomp(R):
Find a set of attributes X s.t.: X<sup>+</sup> ≠ X and X<sup>+</sup> ≠
[all attributes]
```

if (not found) then Return R

<u>let</u> $Y = X^+ - X$, $Z = (X^+)^C$ decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

Return BCNFDecomp(R₁), BCNFDecomp(R₂)

$$\begin{array}{l} \{A\} \rightarrow \{B,C\} \\ \{C\} \rightarrow \{D\} \end{array}$$

Example



Lossless Decompositions



If $\{A_1, ..., A_n\} \rightarrow \{B_1, ..., B_m\}$ Then the decomposition is lossless. $\{A_1, ..., A_n\}$ is a key for one of R1 or R2 Note: don't need { $A_1, ..., A_n$ } \rightarrow { $C_1, ..., C_p$ }

BCNF decomposition is always lossless. Why?

A Problem with BCNF



{Unit} → {Company}
{Company,Product} → {Unit}

We do a BCNF decomposition
on a "bad" FD:
{Unit}+ = {Unit, Company}

{Unit} → {Company}

We lose the FD **{Company, Product}** → **{Unit}**!!

High-Level: Storage and Buffers

- Our model of the computer: Disk vs. RAM
- Buffer Pool
- Replacement Policies

High-level: Disk vs. Main Memory







Disk:

- *Slow:* Sequential access
 - (although fast sequential reads)
- **Durable:** We will assume that once on disk, data is safe!

Random Access Memory (RAM) or Main Memory:

- Fast: Random access, byte addressable
 - ~10x faster for sequential access
 - ~100,000x faster for <u>random access!</u>
- **Volatile:** Data can be lost if e.g. crash occurs, power goes out, etc!
- *Expensive:* For \$100, get 16GB of RAM vs. 2TB of disk!

• Cheap

The Buffer (Pool)

- A <u>buffer</u> is a region of physical memory used to store *temporary data*
 - In this lecture: a region in main memory used to store intermediate data between disk and processes
- *Key idea:* Reading / writing to disk is slowneed to cache data!



Buffer Manager

- Memory divided into **buffer frames**: slots for holding disk pages
- Bookkeeping per frame:
 - *Pin count* : # users of the page in the frame
 - *Pinning* : Indicate that the page is in use
 - Unpinning : Release the page, and also indicate if the page is dirtied
 - **Dirty bit** : Indicates if changes must be propagated to disk

Buffer Manager

- When a Page is requested:
 - In buffer pool -> return a handle to the frame. Done!
 - Increment the pin count
 - Not in the buffer pool:
 - Choose a frame for *replacement*

(Only replace pages with pin count == 0)

- If frame is dirty, write it to disk
- Read requested page into chosen frame
- Pin the page and return its address

Buffer Manager

- When a Page is requested:
 - In buffer pool -> return a handle to the frame. Done!
 - Increment the pin count
 - Not in the buffer pool:
 - Choose a frame for *replacement*

(Only replace pages with pin count == 0)

- If frame is dirty, write it to disk
- Read requested page into chosen frame
- Pin the page and return its address
Buffer replacement policy

- How do we choose a frame for replacement?
 - LRU (Least Recently Used)
 - Clock
 - MRU (Most Recently Used)
 - FIFO, random, ...
- The replacement policy has big impact on # of I/O's (depends on the access pattern)

LRU

- uses a queue of pointers to frames that have pin count = 0
- a page request uses frames only from the *head* of the queue
- when a the pin count of a frame goes to 0, it is added to the *end* of the queue

MRU

- uses a stack of pointers to frames that have pin count = 0
- a page request uses frames only from the *top* of the stack
- when a the pin count of a frame goes to 0, it is added to the top of the stack