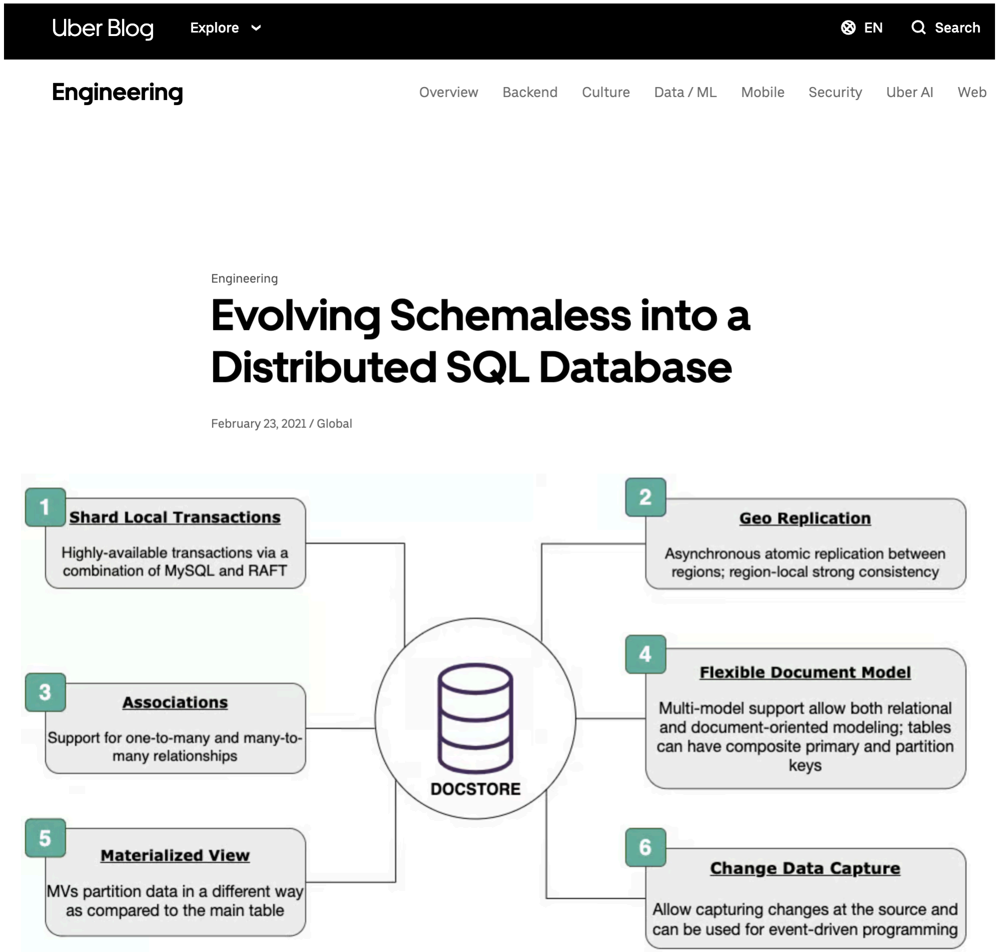


- Flexible schema, not no schema!
- Centralizing the complexity of consistency
- Operational reality may not be "beautiful"

https://www.uber.com/blog/schemaless-sql-database/

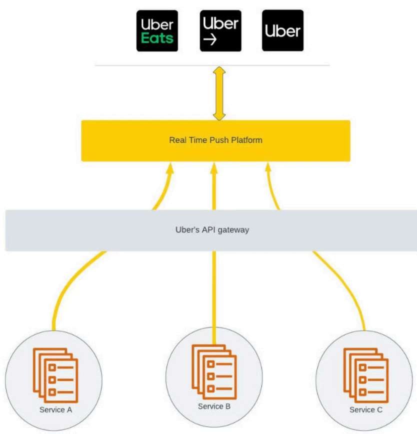
Uber Blog Sign up

Engineering

Engineering, Backend, Mobile

Uber's Next Gen Push Platform on gRPC

August 16, 2022 / Global



In our last blog post we talked about how we went from polling for refreshing the app to a push-based flow to build our app experience.

- Not request/response but streaming
- From REST-like to RPC-like
- Transport is as important as anything else
- Don't reinvent the wheel

https://www.uber.com/blog/ubers-next-gen-push-platform-on-grpc/

Protocol Buffers

- API Schema definition language
- Provides libraries for convenience
- Comparable to JSON and XML
    - But better!
- Binary on the wire

```
edition = "2023";

package tutorial;

message Person {
  string name = 1;
  int32 id = 2;
  string email = 3;

  enum PhoneType {
    PHONE_TYPE_UNSPECIFIED
= 0;
    PHONE_TYPE_MOBILE = 1;
    PHONE_TYPE_HOME = 2;
    PHONE_TYPE_WORK = 3;
  }

  message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
  }

  repeated PhoneNumber
phones = 4;
}

message AddressBook {
  repeated Person people =
1;
}
```

JSON vs. Protobuf

- Example person record in JSON (105 bytes minified)

```
{"name":"Alice Smith","id":12345,"email":"alice@example.com","phones":
[{"number":"555-1234","type":1},{"number":"555-5678","type":2}]}
```

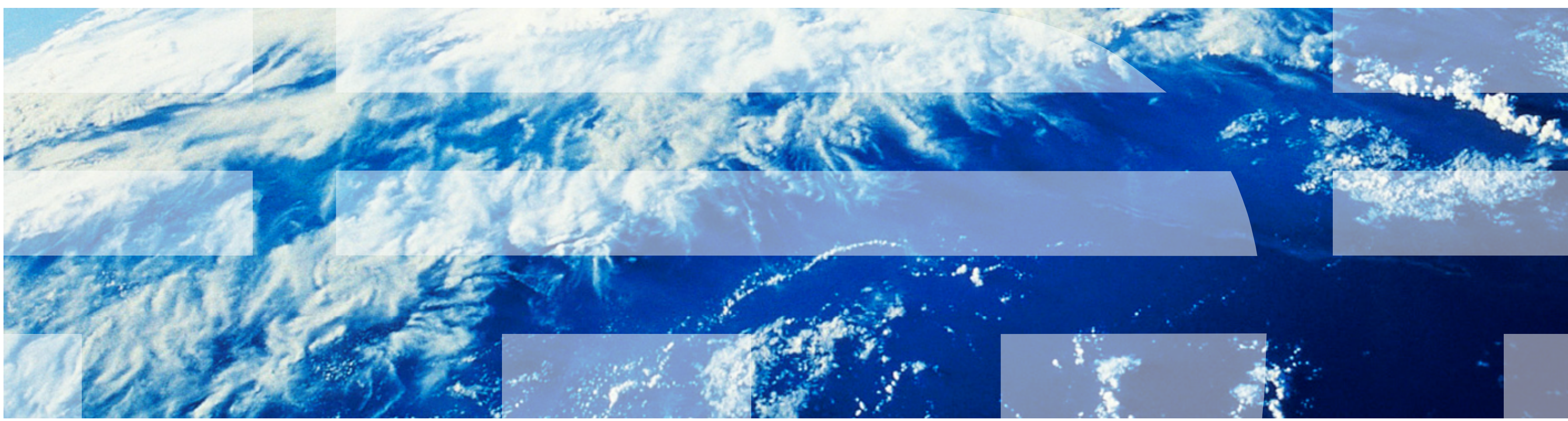
- Same record in Protobuf (45 bytes)

```
0a 0b 41 6c 69 63 65 20 53 6d 69 74 68 10 b9 60 1a 11 61 6c 69 63 65 ...
```

- Why Is Protobuf Smaller?

	JSON	Protobuf
Field names	Repeated as strings ("name", "id")	Integer tags (1, 2, 3...)
Integers	ASCII digits (12345 = 5 bytes)	Varint encoding (2 bytes)
Schema required	No (self-describing)	Yes
Human readable	Yes	No

Single Table Query



SQL Query

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

SELECT <attributes>

FROM <one or more relations>

WHERE <conditions>

Call this a **SFW** query.

Simple SQL Query: Selection

Selection is the operation of filtering a relation's tuples on some condition

PName	Price	Category	Manuf
Gizmo	$19.99	Gadgets	GWorks
Powergizmo	$29.99	Gadgets	GWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT *
FROM   Product
WHERE  Category = 'Gadgets'
```

PName	Price	Category	Manuf
Gizmo	$19.99	Gadgets	GWorks
Powergizmo	$29.99	Gadgets	GWorks

Simple SQL Query: Projection

Projection is the operation of producing an output table with tuples that have a subset of their prior attributes

```
SELECT Pname, Price, Manufacturer
FROM   Product
WHERE  Category = 'Gadgets'
```

PName	Price	Category	Manuf
Gizmo	$19.99	Gadgets	GWorks
Powergizmo	$29.99	Gadgets	GWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

PName	Price	Manuf
Gizmo	$19.99	GWorks
Powergizmo	$29.99	GWorks

Notation

Input Schema

Product(PName, Price, Category, Manufacturer)

```
SELECT Pname, Price, Manufacturer
FROM   Product
WHERE  Category = 'Gadgets'
```

Output Schema

Answer(PName, Price, Manfacturer)

A Few Details

- SQL **commands** are case insensitive:

  Same: SELECT, Select, select

  Same: Product, product

- **Values** are **not:**

  <u>Different:</u> ‘Seattle’, ‘seattle’

- Use single quotes for constants:

  ‘abc’ - yes

  “abc” - no

LIKE: Simple String Pattern Matching

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

- s **LIKE** p: pattern matching on strings
- p may contain two special symbols:
  - % = any sequence of characters
  - _ = any single character

DISTINCT: Eliminating Duplicates

```
SELECT DISTINCT Category
FROM   Product
```

Category
Gadgets
Photography
Household

Versus

```
SELECT Category
FROM   Product
```

Category
Gadgets
Gadgets
Photography
Household

ORDER BY: Sorting the Results

```
SELECT     PName, Price, Manufacturer
FROM       Product
WHERE      Category='gizmo' AND Price > 50
ORDER BY   Price, PName
```

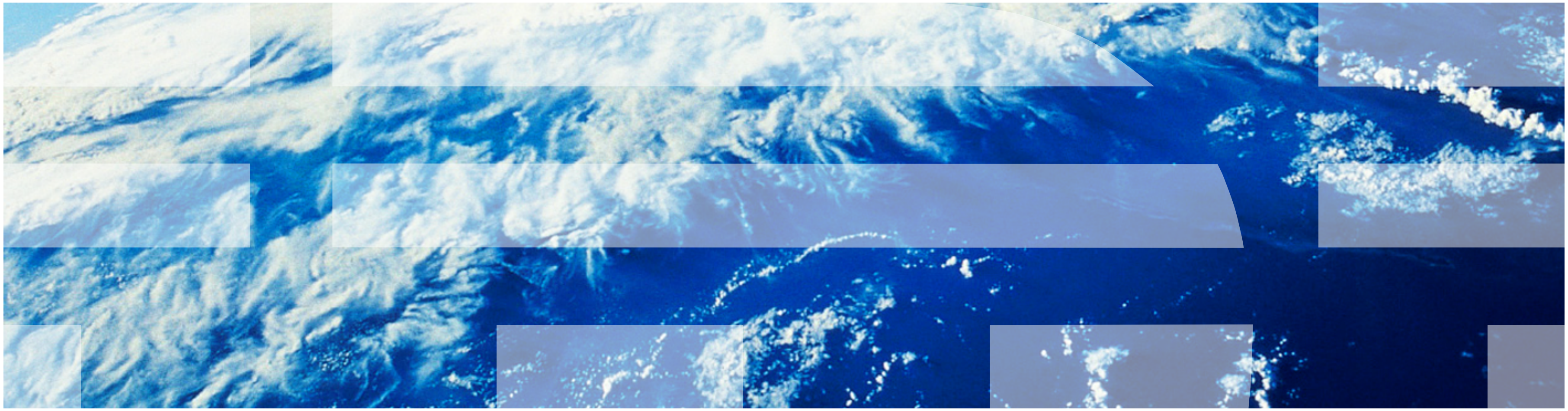
Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.

LIMIT

```
SELECT    PName, Price, Manufacturer
FROM      Product
WHERE     Category='gizmo' AND Price > 50
ORDER BY  Price, PName
LIMIT     5
```

Multi-Table Query



Foreign Key constraints

- Suppose we have the following schema :

> Students(<u>cuid</u>: *string*, name: *string*, gpa: *float*)
>
> Enrolled(<u>student_id</u>: *string*, <u>cid</u>: *string*, grade: *string*)

- And we want to impose the following constraint:

<u>Only bona fide students may enroll in courses'</u> i.e. a student must appear in the Students table to enroll in a class

Students

cuid	name	gpa
102	Bob	3.9
123	Mary	3.8

Enrolled

student_id	cid	grade
102	CS1	A
123	CS4	A+

We say that cuid is a <u>foreign key</u> that refers to Students

Declaring Foreign Keys

Students(<u>cuid</u>: *string,* name: *string,* gpa: *float*)
Enrolled(student_id: *string*, cid: *string,* grade: *string*)

```
CREATE TABLE Enrolled (
  student_id CHAR(20),
  cid  CHAR(20),
  grade  CHAR(10),
  PRIMARY KEY (student_id, cid),
  FOREIGN KEY (student_id) REFERENCES Students(cuid)
)
```

Foreign Keys and update operations

Students(cuid*: string,* name: *string*, gpa: *float*)

Enrolled(student_id: *string,* cid: *string*, grade: *string*)

- What if we insert a tuple into Enrolled, but no corresponding student?

  INSERT is rejected (foreign keys are constraints)!

- What if we delete a student?

  1. Disallow the delete
  2. Remove all of the courses for that student
  3. *SQL allows a third via NULL*

DBA chooses

Keys and Foreign Keys

Company

<u>CName</u>	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

What is a foreign key vs. a key here?

Product

<u>PName</u>	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

shlomi-noach on Dec 9, 2016 · edited by shlomi-noach Edits Contributor ···

At GitHub we do not use foreign keys, ever, anywhere.

Personally, it took me quite a few years to make up my mind about whether foreign keys are good or evil, and for the past 3 years I'm in the unchanging strong opinion that foreign keys should not be used. Main reasons are:

- FKs are in your way to shard your database. Your app is accustomed to rely on FK to maintain integrity, instead of doing it on its own. It may even rely on FK to cascade deletes (shudder).
  When eventually you want to shard or extract data out, you need to change & test the app to an unknown extent.

- FKs are a performance impact. The fact they require indexes is likely fine, since those indexes are needed anyhow. But the lookup made for each `insert` / `delete` is an overhead.

- FKs don't work well with online schema migrations.

This last bullet is not a chicken and an egg, as you might think. FKs impose a lot of constraints on what's possible and what's not possible.

Here's an old post of mine, reviewing the first appearance of Facebook's OSC, and which includes some thoughts on foreign keys: http://code.openark.org/blog/mysql/mk-schema-change-check-out-ideas-from-oak-online-alter-table

Let's say you have two tables, P & C, standing for Parent & Child, respectively. There's a foreign key in C such that each row in C points to some "parent" value in P.

Doing schema migration of C is possible. However since foreign keys have unique names, the new (migrated) C table will have a FK with a different name than the original one.

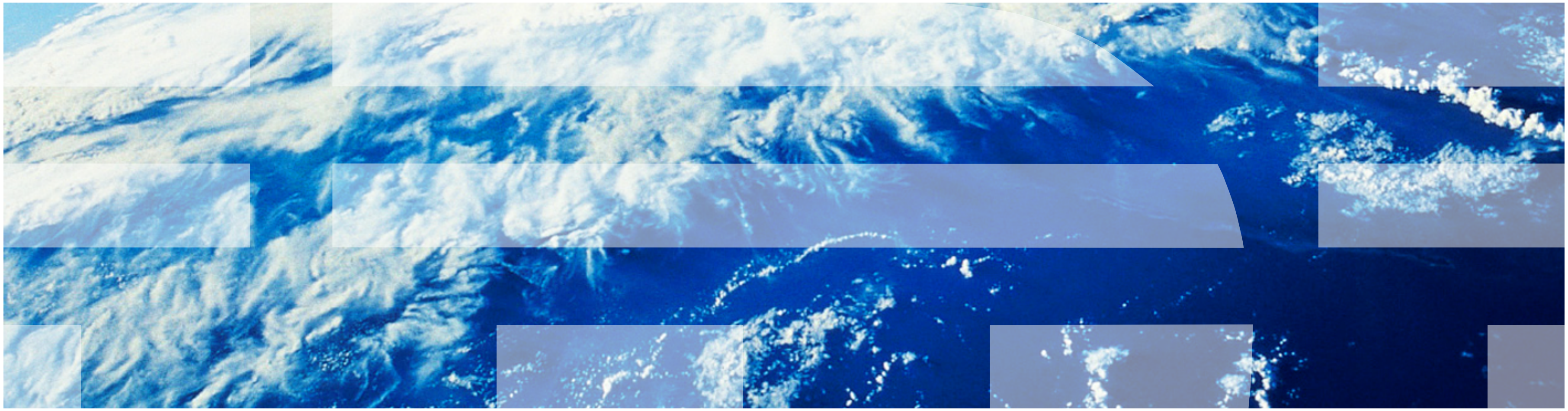
Doing schema migration of P is just not going to work. Recall that `gh-ost` renames the table at the end. Alas, when renaming a table away, the FK will move with the renamed table. To create a parent-side FK on the *ghost* table, one would need to migrate C ; and because `gh-ost` uses async approach, P and P-ghost are never in complete sync at any point in time (except at lock time) which makes it impossible for C to have both a FK to P and to P-ghost. some integrity will be broken.

There's more discussion on the documentation of pt-online-schema-change

👍 290 👎 60 🎉 8 😕 22 ❤️ 60 🚀 11 👀 46

https://github.com/github/gh-ost/issues/331

JOINs and Aggregations



Trade off between table complexity and query complexity

Students(<u>cuid</u>*: string,* name: *string*, gpa: *float*)

Enrolled(<u>student_id</u>: *string,* <u>cid</u>: *string*, grade: *string*)

- What is the GPA of all students enrolled in CSEE 4121?

- A possible (cumbersome solution) → create a new franken-table
  - A single attribute for each possible class:

FrankenTable(<u>student_id</u>: *string,* grade_course1: *string, grade*_course2: *string, …)*

  - Hundreds of attributes, most columns are NULL

Joins

Product(<u>PName</u>, Price, Category, Manufacturer)

Company(<u>CName</u>, StockPrice, Country)

Ex: Find all products under $200 manufactured in Japan; return product names and prices.

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

A <u>**join**</u> between tables returns all unique combinations of their tuples **which meet some specified join condition**

Joins

Product

PName	Price	Category	Manufacturer
Gizmo	$19	Gadgets	GizmoWorks
Powergizmo	$29	Gadgets	GizmoWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

Company

CName	Stock Price	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

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

PName	Price
SingleTouch	$149

An example of SQL semantics

```
SELECT R.A
FROM   R, S
WHERE  R.A = S.B
```

Input

R

A
1
3

S

B	C
2	3
3	4
3	5

Cross Product

A	B	C
1	2	3
1	3	4
1	3	5
3	2	3
3	3	4
3	3	5

Apply Selections / Conditions

A	B	C
3	3	4
3	3	5

Apply Projection

Output

A
3
3

Note: this is how SQL logically works, not actually how it’s implemented

- The preceding slide show *what a join means*
- Not actually how the database executes it under the covers

Aggregations

Product

PName	Price	Category	Manufacturer
Gizmo	$10	Gadgets	GizmoWorks
Powergizmo	$20	Gadgets	GizmoWorks
SingleTouch	$10	Photography	Canon
MultiTouch	$203	Household	Hitachi

```
SELECT AVG(price)
FROM   Product
WHERE  Manufacturer = “GizmoWorks”
```

Output: $15

```
SELECT COUNT(*)
FROM   Product
WHERE  Price > 15
```

Output: 2

- SQL supports several **aggregation** operations:
  - SUM, COUNT, MIN, MAX, AVG

•All operators ignore NULL, **except COUNT**

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  product = 'bagel'
```

⇨ 1*20 + 1.50*20 = 50

Grouping and Aggregation

Purchase(product, date, price, quantity)

```
SELECT   product,
             SUM(price * quantity) AS TotalSales
FROM      Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Find total sales after 10/1/2005 per product.

Let's see what this means…

Grouping and Aggregation

```
SELECT   product,
               SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

<u>Semantics of the query:</u>

1. Compute the FROM and WHERE clauses

2. Group by the attributes in the GROUP BY

3. Compute the SELECT clause: grouped attributes and aggregates

1. Compute the FROM and WHERE clauses

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

FROM

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

2. Group by the attributes in the **GROUP BY**

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

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

GROUP BY

<table>
<tr><th>Product</th><th>Date</th><th>Price</th><th>Quantity</th></tr>
<tr><td rowspan="2">Bagel</td><td>10/21</td><td>1</td><td>20</td></tr>
<tr><td>10/25</td><td>1.50</td><td>20</td></tr>
<tr><td rowspan="2">Banana</td><td>10/3</td><td>0.5</td><td>10</td></tr>
<tr><td>10/10</td><td>1</td><td>10</td></tr>
</table>

3. Compute the SELECT clause: grouped attributes and aggregates

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

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

SELECT →

Product	TotalSales
Bagel	50
Banana	15

HAVING Clause

```
SELECT   product, SUM(price*quantity)
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
HAVING   SUM(quantity) > 100
```

Same query as before, except that we consider only products that have more than
100 buyers

HAVING clauses contains conditions on **aggregates**

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

RECAP: Joins

By default, joins in SQL are **“inner joins”**:

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

```
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM   Product
  JOIN Purchase ON Product.name = Purchase.prodName
```

Both equivalent:
Both INNER JOINS!

Outer Joins

- An **outer join** returns tuples from the joined relations that don't have a corresponding tuple in the other relations
  - I.e. If we join relations A and B on a.X = b.X, and there is an entry in A with X=5, but none in B with X=5…
  - A LEFT OUTER JOIN will return a tuple (a, NULL)!

- Left outer joins in SQL:

```
SELECT Product.name, Purchase.store
FROM   Product
  LEFT OUTER JOIN Purchase ON
             Product.name = Purchase.prodName
```

Now we'll get products even if they didn't sell

INNER JOIN

Product

name	category
iPhone	media
Tesla	car
Ford Pinto	car

Purchase

prodName	store
iPhone	Apple store
Tesla	Dealer
iPhone	Apple store

```
SELECT Product.name, Purchase.store
FROM   Product
 INNER JOIN Purchase
                  ON Product.name = Purchase.prodName
```

name	store
iPhone	Apple store
iPhone	Apple store
Tesla	Dealer

Note: another equivalent way to write an INNER JOIN!

LEFT OUTER JOIN

Product

name	category
iPhone	media
Tesla	car
Ford Pinto	car

Purchase

prodName	store
iPhone	Apple store
Tesla	Dealer
iPhone	Apple store

```
SELECT Product.name, Purchase.store
FROM   Product
  LEFT OUTER JOIN Purchase
                  ON Product.name = Purchase.prodName
```

name	store
iPhone	Apple store
iPhone	Apple store
Tesla	Dealer
Ford Pinto	NULL

Other Outer Joins

- Left outer join:
  - Include the left tuple even if there's no match
- Right outer join:
  - Include the right tuple even if there's no match
- Full outer join:
  - Include the both left and right tuples even if there's no match

How many entries will output table have?

- Left table has L entries
- Right table has R entries

- Inner join:
  - Minimum number of entries: 0
  - Maximum number of entries: L*R

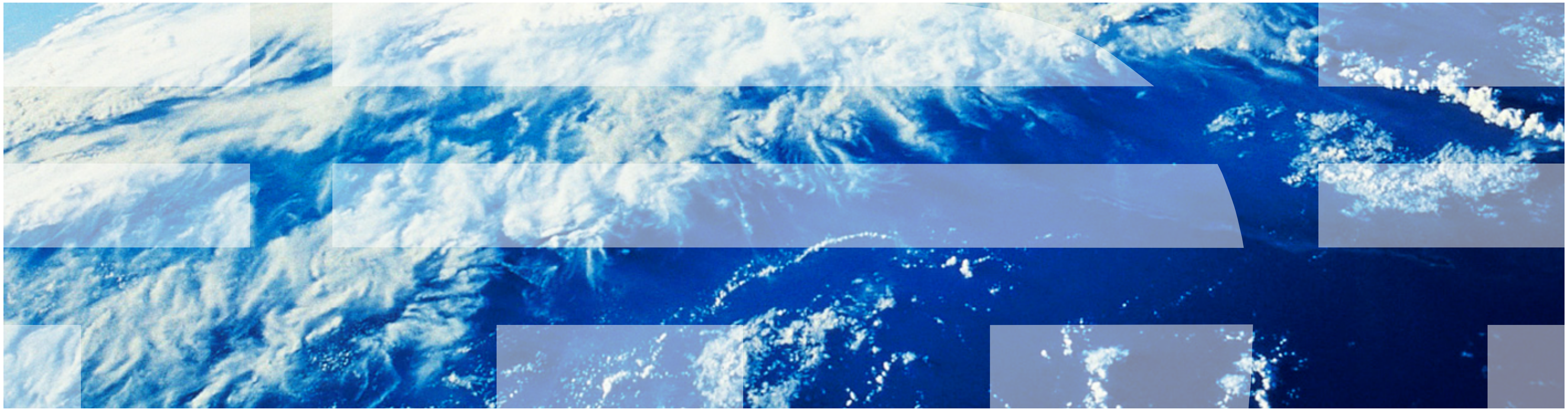
- Left outer join:
  - Minimum number of entries: L
  - Maximum number of entries: L*R

- Right outer join:
  - Minimum number of entries: R
  - Maximum number of entries: L*R

- Full outer join:
  - Minimum number of entries: L+R
  - Maximum number of entries: L*R

Borrowed from Shiva Shivakumar and Theodoros Rekatsinas

Nested Queries



SQL is Compositional

Can construct powerful query chains (e.g., f(g(...(x)))

Inputs / outputs are multisets

⇒ Output of one query can be input to another (nesting)!

⇒ Including on same table

Nested queries: Sub-queries Return Relations

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

```
SELECT   Product.manufacturer
FROM     Purchase, Product
WHERE    Purchase.product = Product.name
         AND Purchase.buyer = 'Alice'`
```

1. Companies making products bought by 'Alice'
2. Location of companies?

Nested queries: Sub-queries Return Relations

Company(name, city)
Product(name, manufacturer)
Purchase(id, product, buyer)

```
SELECT Company.city
FROM   Company
WHERE  Company.name  IN (
       SELECT  Product.manufacturer
       FROM    Purchase, Product
       WHERE   Purchase.product = Product.name
               AND Purchase.buyer = ‘Alice‘)
```

1. Companies making products bought by ‘Alice’
2. Location of companies?

Subqueries Return Relations

You can also use operations of the form:

- s > ALL R
- s < ANY R
- EXISTS R

Example:

Product(name, price, category, maker)

```
SELECT name
FROM   Product
WHERE  price > ALL(
    SELECT price
    FROM   Product
    WHERE  maker = 'Gizmo-Works')
```

Find products that are more expensive than all those produced by “Gizmo-Works”

```
SELECT p1.name
FROM   Product AS p1
WHERE  p1.maker = 'Gizmo-Works'
  AND EXISTS(
         SELECT p2.name
         FROM   Product AS p2
         WHERE  p2.maker <> 'Gizmo-Works'
                AND p1.name = p2.name)
```

<> means !=

Find ‘copycat’ products, i.e. products made by competitors with the same names as products made by “Gizmo-Works”

Note the scoping of the variables!

Example: Complex Correlated Query

Product(name, price, category, maker, year)

```
SELECT DISTINCT  x.name, x.maker
FROM             Product AS x
WHERE            x.price > ALL(
        SELECT y.price
        FROM     Product AS y
        WHERE  x.maker = y.maker
AND y.year < 1972)
```

Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

Can be very powerful (also much harder to optimize)

Aggregates inside nested queries

1. Aggregates inside nested queries. Remember SQL is **compositional**
2. Hint 1: Break down query description to steps (subproblems)
3. Hint 2: Whenever in doubt always go back to the definition

Aggregates inside nested queries: example

Precipitation

station_id	day	precipitation
122	1	33
122	4	20
351	1	10
191	7	45

Example:

"Using a *single SQL query*, find all of the stations that had the highest daily precipitation (across all stations) on any given day."

```
SELECT station_id, day
FROM   Precipitation,
         (SELECT day AS maxd, MAX(precipitation) AS maxp
          FROM precipitation
          GROUP BY day)
WHERE day = maxd AND precipitation = maxp
```

Step 1

```
[redacted]
          (SELECT day AS maxd, MAX(precipitation) AS maxp
           FROM precipitation
           GROUP BY day)
[redacted]
```

maxd	maxp
1	33
4	20
7	45

Step 2

```sql
SELECT station_id, day
FROM   precipitation,
         (SELECT day AS maxd, MAX(precipitation) AS maxp
          FROM precipitation
          GROUP BY day)
WHERE day = maxd AND precipitation = maxp
```

station_id	day	precipitation
122	1	33
122	4	20
351	1	10
191	7	45

JOIN

maxd	maxp
1	33
4	20
7	45

station_id	day
122	1
122	4
191	7

JANUARY 29, 2020

Postmortem: Service disruption on January 21-22, 2020

John Firebaugh Lead Infrastructure Engineer, Figma

INSIDE FIGMA ENGINEERING

The root cause of our recent service outage and our next steps

TL;DR: Why did it happen?

Tuesday's incident was primarily caused by a long-running, expensive query. To prevent this issue from recurring, we will be improving our monitoring of expensive queries and setting tighter bounds on allowed running time.

The root cause of Wednesday's incident was that a routine change in database statistics caused PostgreSQL 9 to mis-plan the execution of one of our queries, causing expensive table scans and writes to temporary buffers. This was exacerbated by concurrent aggressive autovacuuming operations, which were happening because of a vacuuming backlog from the long-running query that had been terminated on Tuesday.

The upgrade to PostgreSQL 11 mitigated both of these causes: improvements to the query planner eliminated the possibility of the bad query plan, and the performance characteristics of autovacuuming have been significantly improved between versions 9 and 11.

https://www.figma.com/blog/post-mortem-service-disruption-on-january-21-22-2020/