Chapter 11

Database Internals: Performance Evaluation

Contents

1 Normalization

- Up to this point we have taken the data in a database as a given without consideration as to what a database *should* look like. In this section we will introduce some of the major concepts around these database design decisions, starting with the process of normalization.
- To motivate this, we need to go back in time and consider one of the common critiques of the relational system originally proposed by Codd, which was that when data within a database is changed there were possible negative side effects. The database normalization process is a set of rules, which if done correctly, will limit the likelihood of this these issues, called *anomalies* occurring.
  1. **Deletion Anomaly:** Non-database information lost due to deleting data within the database.
  2. **Insertion Anomaly:** Incomplete data may mean that we cannot insert information while keeping the database consistent.
  3. **Update Anomaly:** When the database is updated, multiple updates may be required to maintain consistency.
- In order to understand these issues, consider the following tables describing faculty at a school.

Figure 11.1: *faculty* table

FID	Faculty	HireDt	Dept	Dean1	Dean2	Mentor
1	Hamrick	8/95	Finance	Big Boss	Little Boss	Afraid
2	Ross	2/12	Accounting	Small Boss	Medium Boss	Uminsky
3	Parr	2/85	CS	Medium Boss	Small Boss	Hamrick
4	Uminsky	8/11	Math	Big Boss	Biggest Boss	Tao

- Using the table above, how do we write a query which lists all the departments?

```
select distinct dept from faculty;
```

- Using the table above, how do we write a query which returns the number of faculty?

```
select count(*) from faculty;
```

- What can go wrong:
  - **Deletion Anomaly:** What happens when the last Ancient Greek Professor retires? The query above will no longer return Ancient Greek as a department – but the department may still exist!
  - **Insertion Anomaly:** We decide to introduce a new major, but have not hired a professor in that department yet. The query above will not reflect the new department. If we add a row with Null professor to add this information the database becomes inconsistent because the second query above no longer returns the expected information.
  - **Update Anomaly:** In order to get more students, Mathematics decides to change the name of their department to "Mathematical Sciences." In order to make this single "fact" change *every* row in the database related to a math professor must change. if something happened in the middle of the update or if some professor misreported their major next year we would have two math departments when only one should exist.
- How do we avoid this? We *normalize* the table. There are a bunch of different criteria for normalization forms. In our case we will go over a few of the most common ($1^{st}, 2^{nd} and 3^{rd}$ normal forms.

Wikipedia currently lists six ordinal normal forms as well as a few named other version, the most commonly mention (in my experience), being Boyce-Codd Normal Form ("BCNF").

- First Normal Form:
  1. All values within a cell are *atomic* – there is not a cell which contains multiple values. For example, there is no column phone numbers which contains multiple phone numbers in the same row.
  2. Each table has a *key* which logically defines a record.
  3. No repeating columns.
- In our example we violate the no repeating columns rule! So we rewrite the table to avoid this:

Figure 11.2: *faculty* table

FID	Faculty	HireDt	Dept	Dean	DeanNo	Mentor
1	Hamrick	8/95	Finance	Big Boss	1	Afraid
1	Hamrick	8/95	Finance	Little Boss	2	Afraid
2	Ross	2/12	Accounting	Small Boss	1	Uminsky
2	Ross	2/12	Accounting	Medium Boss	2	Uminsky
3	Parr	2/85	CS	Medium Boss	1	Hamrick
3	Parr	2/85	CS	Small Boss	2	Hamrick
4	Uminsky	8/11	Math	Big Boss	1	Tao
4	Uminsky	8/11	Math	Biggest Boss	2	Tao

In this table, what logically defines a row is the faculty-dean combination.

- While the table above moves us toward avoiding the anomalies mentioned above (for example, it is now way easier to change a Dean's name), it still is not a great solution.
- 2nd normal form continues this process, by adding an additional constraint:
  1. All the rules from first normal form
  2. No secondary key (or subset of other key) can have a functional dependency on another attribute. Generally this occurs when you have repeating rows of data related to the key.

In order to implement this normalization procedure we will need to create a new table since we cannot have any repeating rows.

FID	Faculty	HireDt	Dept	Mentor
1	Hamrick	8/95	Finance	Afraid
2	Ross	2/12	Accounting	Uminsky
3	Parr	2/85	CS	Hamrick
4	Uminsky	8/11	Math	Tao

FID	Dean	DeanNo
1	Big Boss	1
1	Little Boss	2
2	Small Boss	1
2	Medium Boss	2
3	Medium Boss	1
3	Small Boss	2
4	Big Boss	1
4	Biggest Boss	2

Figure 11.3: Second Normal Form

- However, this doesn't alleviate all possibility of all the anomalies that we have listed. In order to do that we can try to put the database into third normal formal:

  1. All the rules for second normal form
  2. "Facts" should be independent of the key information that they contain XXX

- In order to put this database in third normal form we need to add *a lot* of tables.

FID	Faculty	HireDt
1	Hamrick	8/95
2	Ross	2/12
3	Parr	2/85
4	Uminsky	8/11

FID	MentorFID
1	92
2	4
3	1
4	85

FID	DID	DeanNo
1	1	1
1	2	2
2	3	1
2	4	2
3	4	1
3	3	2
4	1	1
4	5	2

DeptID	DeptName
10	Finance
12	Accounting
13	CS
40	Math

DID	Dean
1	Big Boss
2	Little Boss
3	Small Boss
4	Medium Boss
5	Biggest Boss

FID	DeptID
1	10
2	12
3	13
4	40

Figure 11.4: Third Normal Form

- The positive of putting the database into third normal form is that you minimize the risk of creating one of the anomalies specified above.
- There are, however, a few downsides:

  1. Many, many tables were created.
  2. The additional tables will need additional effort by the database admins (cost)
  3. With the addition of some many tables, queries may require multiple joins. Doing additional joins may put additional stress on the database.

2 Views

- Normalization frequently leads to schema with lots and lots of tables and, with that, lots of joins that get repeated over and over again in queries. Luckily, relational databases have a system for storing queries within the database without replicating the data. This structure is called a *view* and can be considered a stored query that gets accessed like a table.

- For example lets consider the case where we frequently want to look at Motor Homes from Polk county in our Iowa cars table. We could write the following query to get the data:

```
select * from cls.cars where countyname = 'Polk' and vehiclecat = 'Motor Home';
```

but if we were doing this frequently we could store this as a view instead using the following command:

```
CREATE VIEW cls.polk_motor_homes as (select * from cls.cars where countyname = 'Polk' and vehiclecat = 'Motor Home');
```

- Let's take a look at the following query and what it returns:

```
select
        table_name
        , table_schema
        , table_type
from information_schema.tables
where table_name in ('cars', 'columns');

 table_name |    table_schema    | table_type
------------+--------------------+------------
 columns    | information_schema | VIEW
 cars       | cls                | BASE TABLE
```

- The table type for cars is called a "BASE TABLE" and is the standard table type in the database.
- The columns table, on the other hand, is not a table, it is a view. A **view** is a stored query masquerading as a table. If we run the following query and look at the results you can see the underlying query!

```
select
        table_name
        , view_definition
from
        information_schema.views
where table_name = 'columns';

 table_name |                          view_definition
------------+-----------------------------------------------------
 columns    |  SELECT (current_database())::information_schema.s
```

The view_definition has been truncated as the query underlying this view is long. The premise, however, is straightforward: each time the table "columns" is accessed, the query in the view definition is executed.

- We can define a view using any select query. If we are analyzing Motorcycle registrations from Lucas County frequently we could execute the following query:

```
 create view cls.mc_lucas as
         select
                 countyname
                 , year
                 , registrations
         from cls.cars
                 where countyname ='Lucas'
                 and vehicletype = 'Motorcycle';
```

This would create a view which we could easily access using SQL and it will also be in the information_schema:

```
select *
        from mc_lucas
order by year;

 countyname | year | registrations
 -----------+------+---------------
 Lucas      | 2005 |           530
 Lucas      | 2006 |           586
 Lucas      | 2007 |           606
 Lucas      | 2008 |           592
 Lucas      | 2009 |           587
 Lucas      | 2010 |           588
 Lucas      | 2011 |           578
 Lucas      | 2012 |           582
 Lucas      | 2013 |           586


select
        table_name
        ,table_type
from
        information_schema.tables
where table_name = 'mc_lucas';

 table_name | table_type
------------+------------
 mc_lucas   | VIEW
```

- As before, the query defining the view is found in the "views" table:

```
select
        table_name, view_definition
from information_schema.views
where table_name = 'mc_lucas';

-[ RECORD 1 ]---+-----------------------------------------------------
table_name      | mc_lucas
view_definition |  SELECT cars.countyname,                             +
|   cars.year,                                                       +
|   cars.registrations                                               +
|   FROM cls.cars                                                    +
|   WHERE (((cars.countyname)::text = 'Lucas'::text)                 +
|   AND ((cars.vehicletype)::text = 'Motorcycle'::text));
```

As shown above, the original query defining the view has been modified by the database. This modification does not change what the query returns.

- In order to get rid of the view, we use the DROP VIEW command:

```
drop view cls.mc_lucas;
```

At which point in time the view is removed.

- Why are views useful?
  - **Shared Query**: If an entire team is doing a set of analysis on a particular topic, using a view to create a common table can ensure consistency between different team members.
  - **Version Control**: As team members learn about a topic they can use views to keep everyone up-to-date with the latest findings. For example, a table with the name "Fraud" which detects fraudulent transactions can be updated to include new algorithms for detecting fraud without everyone having to re-write their queries.
  - **Laziness:** Using a view means less typing for everyone!
- Downside of views:
  - **Performance considerations:** For a number of reasons, views tend to be less performant than using a real query. In some relational databases, views can act as an *optimization barrier*. Consider the following example:

    ```
    select * from cls.mc_lucas where county = 'Lucas';
    ```

    In this example, the underlying mc_lucas view removes all rows which do not refer to Lucas county. However, in *some* circumstances the view will act as an optimization barrier and check each row twice, once in the view and once in the outer where, to verify that the rows are from Lucas county.
  - **Proliferation of Views:** If everyone on a data team is allowed to create views the tendency is for the number of views to explode. If there are too many views it becomes difficult to find and they go unused.

3 Information Schema

- Returning to Codd and his 4th rule:

  > The database is represented at the logical level the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data

- What does this mean? Reading it closely, note that Codd is saying that *the database* itself is represented the same way as *ordinary data*. Users of the database should be able to use the same relational language (SQL) to get answer questions about the database.
- In other words: Everything about the database is in the database itself.
- Unfortunately, this representation is specific to the SQL variant. Postgres contains database specific information in a variety of places, though the most common place to access it is via the tables in the schema "information_schema"
  - Tables contains information on the tables within the database.
  - Columns contains column specific information
  - Views contains information on views, which we will discuss later.
- For example:

```
select
        *
from
        information_schema.columns
where
        column_name = 'registrations';
```

  Will return all the information about the registrations column from the cars table.

- Detailed information about the columns in each table:

```
select
        column_name
        , data_type
from
        information_schema.columns
where table_name ='cars';

column_name    |     data_type
-------------------+-------------------
year               | integer
yearending         | date
countyname         | character varying
countycode         | integer
motorvehicle       | character varying
vehiclecat         | character varying
vehicletype        | character varying
tonnage            | character varying
registrations      | integer
annualfee          | double precision
primarycountylat   | double precision
primarycountylong  | double precision
primarycountycord  | character varying
```

With this query we can identify the data types associated with each column, which should match the data types used to define the table.

- We can write queries against the information schema using any SQL functionality. For example, we could identify all the character columns with the following:

```
select *
from
        information_schema.columns
where
        data_type like 'char%';
```

- A common question asked by new users of SQL is if it is possible to store additional data about the database within a table. The answer is "Yes" – you can create a table which contains information about the database, but that table will have to updated by hand. There is no automatically generated data dictionary created when using a relational database.

4 Performance Considerations

Before talking about query performance and how to measure it we need to discuss two important questions:

1. How the data is stored on the hard drive:

    - The easiest mental model for how data is stored on a hard drive is to consider the database as a record player, where each row is stored on the hard drive in a random order, back-to-back. In other words, where one row ends another row begins.

- There are a couple of interesting aspects to why this is important. First off, if every row is the same length then finding the start and end of a row can be pretty easy. For example, if every row is 25 bytes long, then going to the fifth row entails simply seeking ahead $25 \cdot 5 = 125$ bytes. If the rows are arbitrarily long, then this process can take some time.
- The above is why there is a difference between char and varchar data types. A char datatype is a *fixed* length while a varchar is of variable length. In some database systems storing data as a char can increase performance significantly.[1]
- In order to find data on a database, the computer must seek around the drive, which is incredibly slow, even on modern databases. Consider the following table which describes the relative speed of different random access levels. [2]

Figure 11.5: Access Speeds

Access type	Actual time	Approximated time
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 μs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years

Looking at this table, you can see that seeking data, even on an SSD, can be incredibly slow in comparison to processing speeds. Fundamentally, a database stores data and attempts to provide access to it as quickly as possibly – so it is always running up against this hurdle.

2. How the database decides how to execute a particular query:
   - Basically, the database takes your SQL query and runs it through a database optimizer which rewrites your query a number of different ways and then estimates how long each way will take. The database will then execute the one that it believes will take the smallest amount of time.
   - The query optimizer can often be wrong.
   - There are a number of different ways that it can be wrong, but most of them relate to "cost" – or how long the database thinks a particular operation will cost. In order to estimate the cost, the database keep statistics on each table and column. For example, consider the following:

[1] This is not true of Postgres, which stores both varchar and char the same way.
[2] Source: `https://madusudanan.com/blog/understanding-postgres-caching-in-depth/`

```
 select * from pg_stats where tablename = 'mta' and attname = 'direction' ;
-[ RECORD 1 ]----------+----------------
schemaname             | cls
tablename              | mta
attname                | direction
inherited              | f
null_frac              | 0
avg_width              | 2
n_distinct             | 2
most_common_vals       | {I,O}
most_common_freqs      | {0.5276,0.4724}
histogram_bounds       |
correlation            | 0.450035
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |
```

This query contains information on the direction column within the MTA table. You can see that there is a host of information about the contents. This includes the frequency information. However:

```
 select count(1), direction from cls.mta group by 2;
-[ RECORD 1 ]-----
count     | 613608
direction | I
-[ RECORD 2 ]-----
count     | 552120
direction | O

ncross=# select 613608.0 / ( 552120 + 613608.0);
-[ RECORD 1 ]--------------------
?column? | 0.52637321913859836943
```

The frequency information is not correct!

- The database itself does not keep perfect statistics on each column, it would take too long and instead runs on estimates that it will occasionally update.

- So how can we know what the database is doing and if it is being efficient?
- Every major relational database system has a query analyzer or "explain" which gives you insight into what the database is doing.
- With Postgres, there are two commands that are used to do this: EXPLAIN and EXPLAIN ANALYZE. Generally speaking these commands do not show up nicely when using an SQL client, using psql will result in better output.
- EXPLAIN will tell you what the query optimizer *thinks* will happen while EXPLAIN ANALYZE will return both what the query optimizer thought would happen and what actually happened.
- Let's look at the following example:

```
ncross=# explain analyze select plaza, sum(hr) from cls.mta where hr = 1 group by 1 having plaza = 2;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=0.00..24937.72 rows=1 width=8) (actual time=149.688..149.688 rows=1 loops=1)
   Group Key: plaza
   ->  Seq Scan on mta  (cost=0.00..24911.92 rows=5159 width=8) (actual time=15.469..148.941 rows=5124 loops=1)
         Filter: ((hr = 1) AND (plaza = 2))
         Rows Removed by Filter: 1160604
 Planning time: 0.074 ms
 Execution time: 149.731 ms
(7 rows)

ncross=# explain analyze select plaza, sum(hr) from cls.mta where plaza = 2 and hr = 1 group by 1;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=0.00..24937.72 rows=1 width=8) (actual time=137.129..137.129 rows=1 loops=1)
   Group Key: plaza
   ->  Seq Scan on mta  (cost=0.00..24911.92 rows=5159 width=8) (actual time=13.293..136.394 rows=5124 loops=1)
         Filter: ((plaza = 2) AND (hr = 1))
         Rows Removed by Filter: 1160604
 Planning time: 0.071 ms
 Execution time: 137.163 ms
(7 rows)
```

- In the above we can see that the query plan is *the same* for the two queries despite the use of the HAVING clause in the first. In other words, the database was smart enough to remove the plazas which were not equal to 2 before doing the aggregation.
- In order to read these you start from the bottom and go up.
- Another one. CTE acting as optimization blocker

```
explain analyze with
        MTALTV as (select
            plaza, mtadt, hr, direction
            , sum(vehiclesez) over(partition by plaza order by mtadt, hr rows between unbounded preceding and current row) as runningS
        from
                cls.mta)
select
        *
from
        MTALTV
where plaza =1 and hr = 0 and mtadt = '2010-01-01' and direction = 'I'

                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 CTE Scan on mtaltv  (cost=186684.07..221655.91 rows=1 width=28) (actual time=898.385..1960.692 rows=1 loops=1)
   Filter: ((plaza = 1) AND (hr = 0) AND (mtadt = '2010-01-01'::date) AND ((direction)::text = 'I'::text))
   Rows Removed by Filter: 1165727
   CTE mtaltv
     ->  WindowAgg  (cost=160455.19..186684.07 rows=1165728 width=22) (actual time=898.361..1541.795 rows=1165728 loops=1)
           ->  Sort  (cost=160455.19..163369.51 rows=1165728 width=18) (actual time=897.422..1058.579 rows=1165728 loops=1)
                 Sort Key: mta.plaza, mta.mtadt, mta.hr
                 Sort Method: external merge  Disk: 34232kB
                 ->  Seq Scan on mta  (cost=0.00..19083.28 rows=1165728 width=18) (actual time=0.022..202.735 rows=1165728 loops=1)
 Planning Time: 0.293 ms
 Execution Time: 1985.602 ms
(11 rows)
```

```
explain analyze with
        MTALTV as (select
            plaza, mtadt, hr, direction
            , sum(vehiclesez) over(partition by plaza
                    order by mtadt, hr rows
                        between unbounded preceding and current row) as runningS
        from
                cls.mta
                where plaza =1 and hr = 0 and mtadt = '2010-01-01' and direction = 'I')
select
        *
from
        MTALTV
limit 100;

                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=27826.27..27826.31 rows=2 width=20) (actual time=19.346..147.565 rows=2 loops=1)
   CTE mtaltv
     ->  WindowAgg  (cost=0.00..27826.27 rows=2 width=16) (actual time=19.342..147.555 rows=2 loops=1)
           ->  Seq Scan on mta  (cost=0.00..27826.24 rows=2 width=16) (actual time=19.314..147.520 rows=2 loops=1)
                 Filter: ((plaza = 1) AND (hr = 0) AND (mtadt = '2010-01-01'::date))
                 Rows Removed by Filter: 1165726
   ->  CTE Scan on mtaltv  (cost=0.00..0.04 rows=2 width=20) (actual time=19.344..147.561 rows=2 loops=1)
 Planning time: 0.108 ms
 Execution time: 147.609 ms
(9 rows)
```

```
 explain analyze select
        plaza, mtadt, hr
        , sum(vehiclesez) over(partition by plaza
            order by mtadt, hr rows
            between unbounded preceding and current row) as runningS
    from
        cls.mta
        where plaza =1 and hr = 0 and mtadt = '2010-01-01' limit 100 ;
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.00..27826.27 rows=2 width=16) (actual time=19.450..137.015 rows=2 loops=1)
   ->  WindowAgg  (cost=0.00..27826.27 rows=2 width=16) (actual time=19.448..137.011 rows=2 loops=1)
         ->  Seq Scan on mta  (cost=0.00..27826.24 rows=2 width=16) (actual time=19.435..136.992 rows=2 loops=1)
               Filter: ((plaza = 1) AND (hr = 0) AND (mtadt = '2010-01-01'::date))
               Rows Removed by Filter: 1165726
 Planning time: 0.116 ms
 Execution time: 137.065 ms
```

- These commands give us a ton of insight into how a query is operating.

5 Index

- So what do we do if a query is slow?
- The easy answer is usually to add an *index*, which is a tree structure which allows for searching for values quickly. Specifically, the goal of an index is to decrease the number of *random* hard drive accesses.

```
explain analyze select retdate, symb, sum(cls) over(partition by symb order by retdate asc) from stocks.s2010 limit 10;

                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=81387.26..81387.46 rows=10 width=40) (actual time=14700.353..14700.428 rows=10 loops=1)
   ->  WindowAgg  (cost=81387.26..92327.78 rows=547026 width=40) (actual time=14700.351..14700.424 rows=10 loops=1)
         ->  Sort  (cost=81387.26..82754.82 rows=547026 width=40) (actual time=14699.988..14700.059 rows=11 loops=1)
               Sort Key: symb, retdate
               Sort Method: external merge  Disk: 23720kB
               ->  Seq Scan on s2010  (cost=0.00..14293.26 rows=547026 width=40) (actual time=0.038..481.515 rows=816066 loops=1)
 Planning time: 1.862 ms
 Execution time: 14715.666 ms
(8 rows)
```

vs.

```
create index tst2 on stocks.s2010 (symb, retdate);

 explain analyze select retdate, symb, sum(cls) over(partition by symb order by retdate asc) from stocks.s2010 limit 10;
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.42..1.31 rows=10 width=16) (actual time=0.388..1.308 rows=10 loops=1)
   ->  WindowAgg  (cost=0.42..72501.81 rows=816066 width=16) (actual time=0.386..1.302 rows=10 loops=1)
         ->  Index Scan using tst2 on s2010  (cost=0.42..58220.66 rows=816066 width=16) (actual time=0.301..1.077 rows=11 loops=1)
 Planning time: 2.509 ms
 Execution time: 1.715 ms
(5 rows)

drop index stocks.tst2;
```

- So why not use Index's everywhere?
- The downside of using index is that they take additional hard drive space and increase the cost of any additional data writes or updates. Since the index has to be kept up to date with the data on disk this means that all writes or updates to any table needs to be reflected in changes to all appropriate indexes.
- There are many different types of indexes and these different types allow for efficiencies for different access methods. Examples include things like B-trees which allow for fast comparison operators

6 Distributed Systems and the CAP Theorem

- In this section we will start talking about *distributed* systems, or multiple computer systems which are networked together to act as a single unit.
- When dealing with these systems, each computer is generally refereed to as a "node" and the entire system called a "cluster."
- Distributed systems work by spreading work around different nodes. For example, if you have a 10-node cluster and send it two queries, those queries will not necessarily land on the same node each time and thus the physical machine returning the result to you might be different.
- Many instances require distributed systems because the amount of data generated is too large to be handled by a single computer.
- Consider the following examples:
  1. **Zynga (2014)**

     500-1000 Node Vertica cluster

     60B rows/day

     10 TB/day
  2. **Sega (2015)**

     8 Node Redshift cluster

     20 TB

     Main Table had 4B rows

     110MM rows/day

     Incredibly spiky data flows
  3. **GSN (2014)**

     200 Node Vertica cluster

Every 15 minutes 5MM rows loaded

Biggest table had 100B rows

- An important mathematical theorem governing these types of systems is called the CAP theorem, which is a non-existence result. The CAP theorem states that no distributed system can be **C**onsistent, **A**vailable and **P**artition tolerant. You can only choose two of these three things. Note that this theorem is a mathematical result and like many such results there is an ongoing discussion of how well the mathematical definitions of these terms line-up with reality. That discussion is beyond the scope of this class.
  - **Consistency:** A system is consistent if all nodes within the system respond the same way to the same query. For example, if you send a query to the first node in a cluster and it responds with data X then that same query, if the second node responded, should also return data X.[3]
  - **Availability:** A system is available if, when one node goes down, the system is still able to respond to questions.
  - **Partition Tolerance:** Nodes will continue to perform even if there is a disruption in communication between them.
- As an example, consider the following story of a startup that you create call 1-800-REMINDME:[4]
  - The basic business model is that people call 1-800-REMINDME and then state their name and something that they want to remember, such as "Jeff, I have a meeting at 8AM."
  - If you call back it will return what you said the first time and charge them 10 cents.
  - **Day #1:** You sit down with your notebook and start writing down and responding to calls. This works great!
  - **Day #2:** TechCrunch and VentureBeat both publish articles about your hot new startup. You are now swamped and when people call they are getting busy signals!
  - **Day #3:** You add your sig-o to help, who sits in another room, on another line, with her own notebook. Unfortunately, this doesn't work! Sometimes people call and get your sig-o, sometimes they call and get you, but either way, some messages end up in their notebook and some in yours:

    ```
    Jeff: When is my meeting?
    sig-o: You don't have a meeting.
    Jeff: Wow. You suck.
    ```

  - This system is not **consistent**, as, depending on what node you contact you get different responses.
  - **Day #4:** New plan! In order to make the system consistent you do the following: before making any create or update operation confirm it with the other person:

    ```
    Jeff to sig-o: My class is at 8AM
    sig-o to you: Jeff, class at 8AM
    you to sig-o: Jeff, class at 8AM confirmed.
    sig-o to Jeff: confirmed.
    ```

[3] Note that this is a different type of consistency then the type of consistency we spoke of when talking about transactions and ACID.

[4] This was taken from `http://ksat.me/a-plain-english-introduction-to-cap-theorem/`

- This slows down the system on writes, but reads are very quick and consistent since both you and your sig-o's notebook are the same.
- **Day #5:** Oh, no, sig-o get sick and sometimes run out of the room to throw-up! What happens then...

```
Jeff to you: My meeting is at 9AM.
you to sig-o: Jeff, meeting at 9AM.
sig-o: ....
Jeff hangs up after 5 minutes
```

- This system is not **available**, when one node goes down the entire system fails.
- **Day #6:** New idea: If the other person is not reachable then write down all writes and send via email to the other person. The other person does not start answering the phone after coming back before checking their email for a list of all changes and verifying that all those writes are in their system.
- This system is now consistent and available!
- **Day #7:** Your sig-o is pretty sick of this business and just stops talking to you, or sending emails or responding at all. Once again you are stuck, because you don't get an email from them you don't know what changes happened. Because you can't confirm any transactions you can't do any updates or add new clients to your notebook.
- This system is not partition tolerant. If there is no communication between the nodes then the system once again fails. Even though there is perfectly good information for many of 1-800-REMINDME in your notebook, you can't use it because the system that you have in place requires verification from the non-responsive sig-o.

- The CAP theorem states that you can only really choose 2 of the 3 CAP elements.
- Different applications will require different attributes. For example, bank account information will always want consistency – account balances should always report the same number.
- Note that there are also minor definitional changes in each of these. For example, many systems have what is called *eventual* consistency. Redshift, one of Amazon's offerings, promises consistency within 60 seconds.