Unstructured Database Design

Business Data Management and Analytics

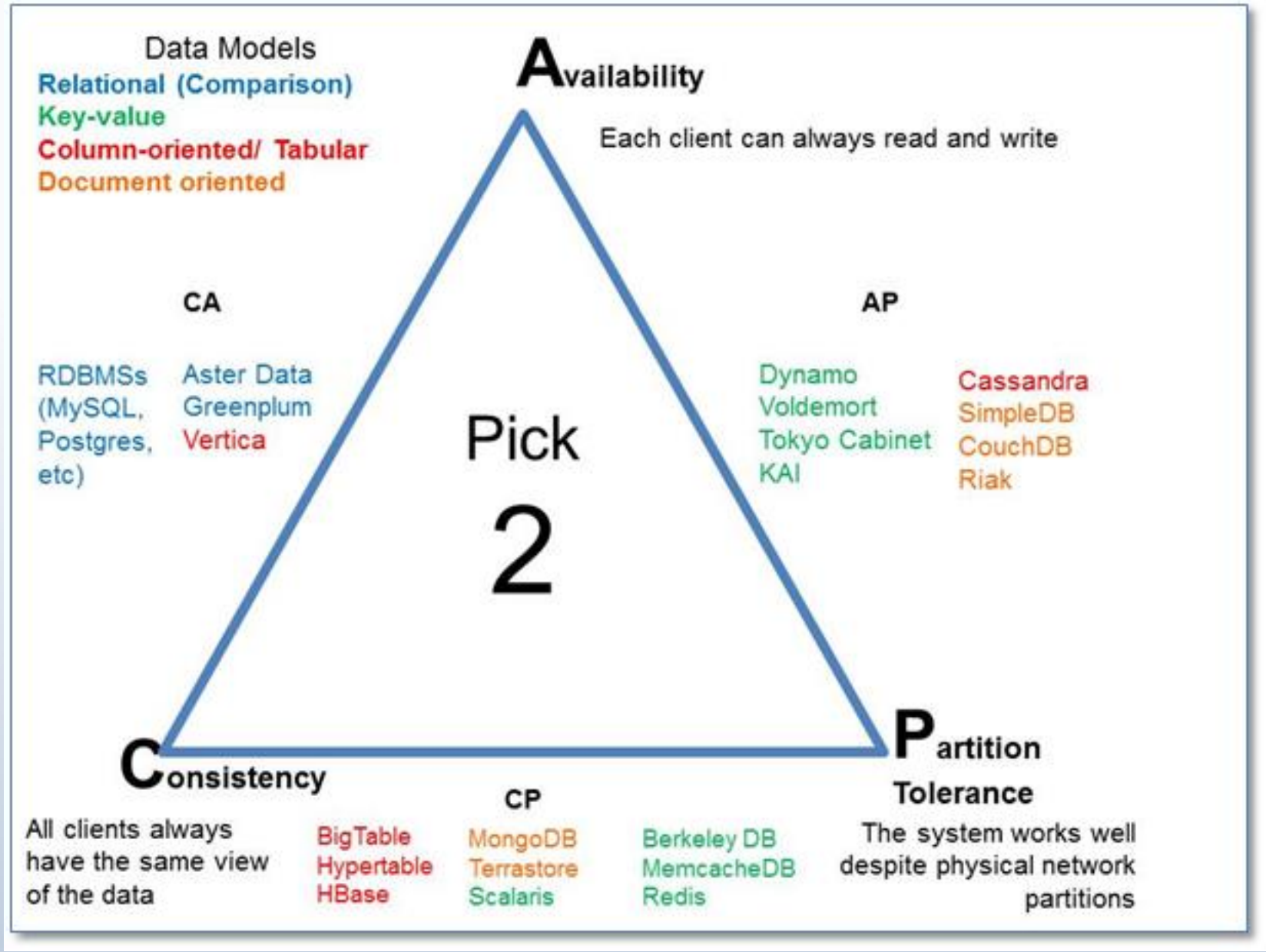
MongoDB

History

- noSQL - (often interpreted as Not Only SQL) database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases.
- mongoDB = "Humongous DB"
  - Open-source
  - Document-based
  - "High performance, high availability"
  - Automatic scaling
  - C-P on CAP

Theories impacting databases

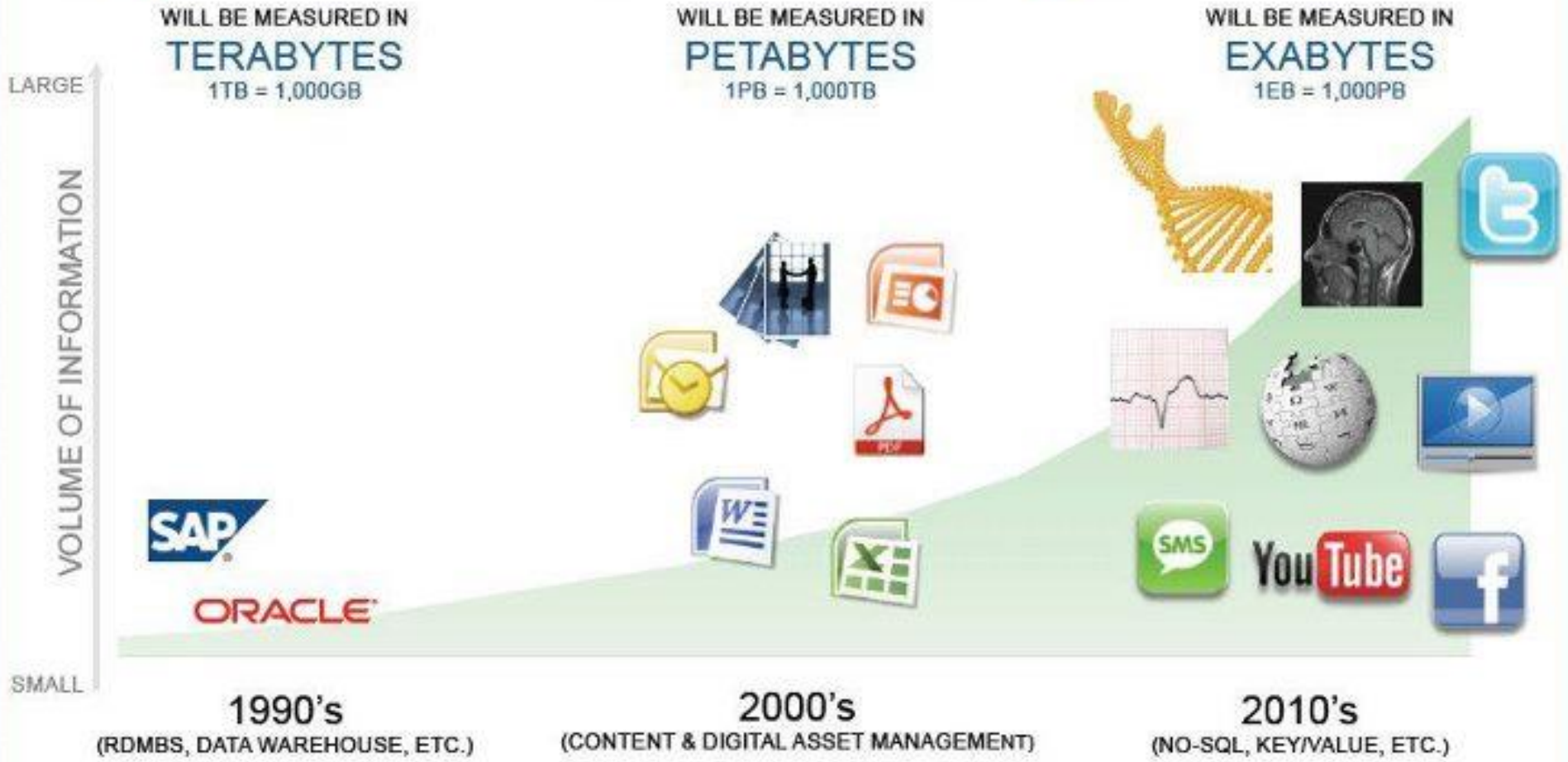
- Relational Model – normalisation
- ACID theorum
- CAP Theorum
- BASE – Basically Available, Soft state and Eventually consistent


Data Models
Relational (Comparison)
Key-value
Column-oriented/ Tabular
Document oriented
Availability
Each client can always read and write
CA
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
Pick 2
AP
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Consistency
All clients always have the same view of the data
CP
BigTable
Hypertable
HBase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis
Partition Tolerance
The system works well despite physical network partitions

Why NoSQL?

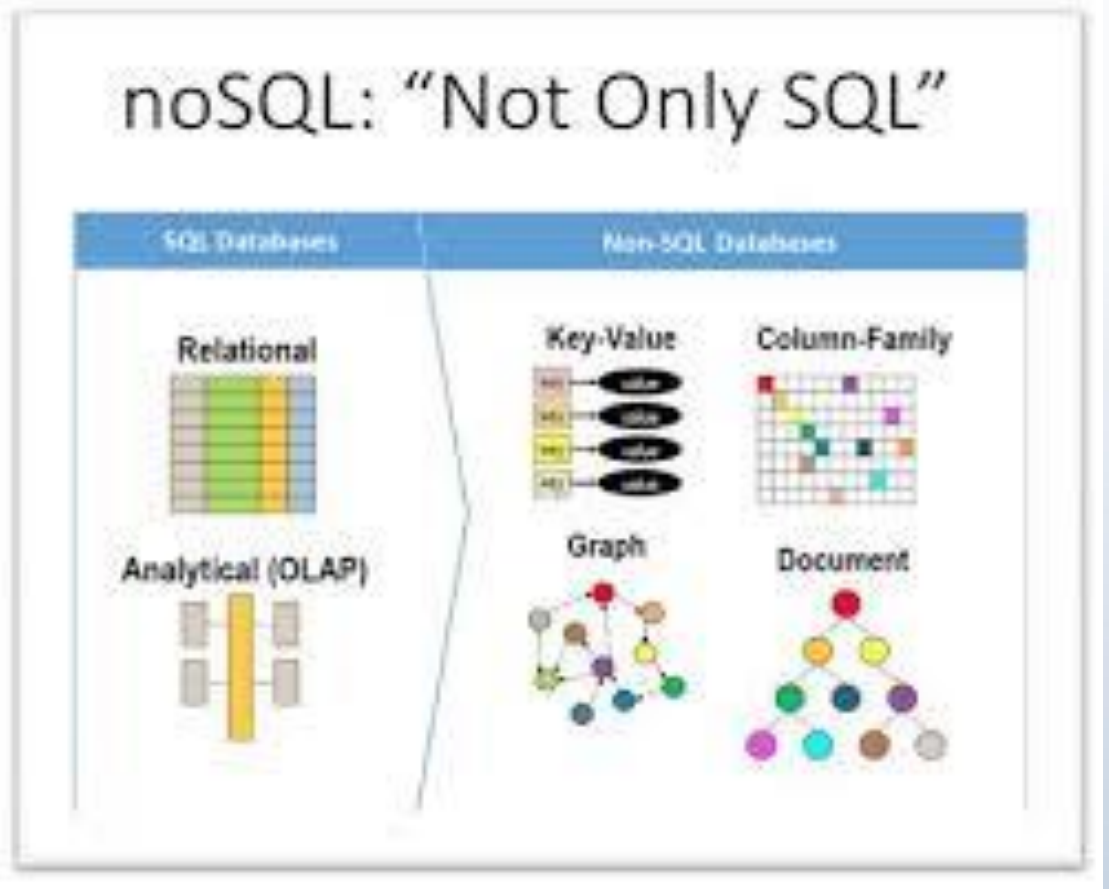
- Big Users – Global usage, increased daily use, mobile
- Big Data – Social data, multimedia, mapping/location, shopping, advertising, communication, entertainment, etc.
- Cloud Computing – infrastructure to support
- NoSQL hopes to support larger numbers of users, a data model that maps the growing needs of users and data, and supports the infrastructure requirements.

Big Data Will Scale To Exabytes



Other NoSQL Types

- Wide Column Store
- Document Store
- Key Value/Tuple Store
- Graph Database
- MultiModal Database
- Object Databases
- Grid & Cloud Solutions
- XML Database
- Multidimensional/Multivalue databases



http://nosql-database.org/


no:sql
talis
apache
CouchDB
relax
Cassandra
Neo4j
the graph database
Apache
Solr
mongoDB
{name: "mongo", type: "db"}
Redis
SQL
ORACLE
PostgreSQL
Microsoft
SQL Server
MySQL
SQLite

Motivations

- Problems with SQL
  - Rigid schema
  - Not easily scalable (designed for 90's technology or worse)
  - Requires unintuitive joins
- Perks of mongoDB
  - Easy interface with common languages (Java, Javascript, PHP, etc.)
  - DB technology should run anywhere (VM's, cloud, etc.)
  - Keeps essential features of RDBMS's while learning from key-value noSQL systems

In Good Company

PEARSON

MUSIC TELEVISION

LinkedIn

Expedia

sourceforge

Data Model

- Document-Based (max 16 MB)
- Documents are in BSON format, consisting of field-value pairs
- Each document stored in a collection
- Collections
  - Have index set in common
  - Like tables of relational db's.
  - Documents do not have to have uniform structure

BSON Example

```
{
        "_id" : 1,
        "name" : { "first" : "John",
                    "last" : "Backus"
                  },
        "contribs" : [ "Fortran", "ALGOL", "Backus-Naur Form", "FP" ],
        "awards" : [
                    {
                      "award" : "W.W. McDowell Award",
                      "year" : 1967,
                      "by" : "IEEE Computer Society"
                    },
                    { "award" : "Draper Prize",
                      "year" : 1993,
                      "by" : "National Academy of Engineering"
                    }
                  ]
}
```

The _id Field

- By default, each document contains an _id field. This field has a number of special characteristics:
  - Value serves as primary key for collection.
  - Value is unique, immutable, and may be any non-array type.
  - Default data type is ObjectId, which is "small, likely unique, fast to generate, and ordered." Sorting on an ObjectId value is roughly equivalent to sorting on creation time.

mongoDB vs. SQL

mongoDB	SQL
Document	Tuple/Record/Row
Collection	Table/View/Relation
PK: _id Field	PK: Any Attribute(s)
Uniformity not Required	Uniform Relation Schema
Index	Index
Embedded Structure	Joins
Shard	Partition

CRUD SQL: Create/Insert

- To insert documents into collection/make a new collection:

mongoDB queries	SQL queries
db.plan.insert({ planname: "Yes10", connectfee: 1.00, peakfee: 1.05, offpeakfee: 0.90, weekendfee: 0.85 });	INSERT INTO plan (PlanName, ConnectFee, PeakFee, OffPeakFee, WeekendFee) VALUES ('Yes10', 1.00, 1.05, 0.90, 0.85);
OR instead of planname: _id: "Yes10"	Note: above assumes CREATE TABLE plan (PlanName varchar(20) PRIMARY KEY, ConnectFee decimal(8,2) , PeakFee decimal(8,2), OffPeakFee decimal(8,2), WeekendFee decimal(8,2));

- To insert multiple documents, use an array.

CRUD SQL: Querying

- Done on collections.
  - Add .limit(<number>) to limit results

mongoDB queries	SQL queries
db.plan.find()	SELECT * FROM plan;
db.plan.find({planname:"Yes20"})	SELECT * FROM plan WHERE planname = "Yes20";
db.plan.find({ planname: "Freestyle", peakfee: 2 });	SELECT * FROM plan WHERE planname = "Freestyle" AND peakfee =2;
db.plan.find({ "$or": [{ planname: "Yes10" }, { planname: "Yes20" }] });	SELECT * FROM plan WHERE planname = "Yes10" OR planname = "Yes20";

CRUD SQL: Querying

mongoDB queries	SQL queries
db.plan.find({ planname: "Yes20", connectfee: { "$gt": 1.5 } });	SELECT * FROM plan WHERE planname = "Yes20" AND connectfee > 1.5;
db.plan.find({ planname: {"$in": ["Yes10", "Yes20"]} });	SELECT * FROM plan WHERE planname in ("Yes10","Yes20");
db.plan.find({}, { planname: 1, connectfee: 1 });	SELECT planname, connectfee FROM plan;
db.plan.find({ peakfee: 1.35 }, { planname: 1, connectfee: 1 });	SELECT planname, connectfee FROM plan WHERE peakfee = 1.35;

- Find documents with or w/o field:
  db.plan.find({connectfee: { $exists: true}})

CRUD: Updating

mongoDB queries	SQL queries
db.plan.update({planname:"Yes10"}, {$set: {connectfee:2.20}}, {multi:true}) //update multiple docs	UPDATE plan SET connectfee = 2.20 WHERE planname = "Yes10";

- upsert: if true, creates a new doc when none matches search criteria.

CRUD: Removal

mongoDB queries	SQL queries
db.plan.remove({planname:"Yes40"})	DELETE FROM plan WHERE planname = "Yes40";

- As above, but only remove first document

db.<collection>.remove({<field>:<value>}, true)

Mongo is basically schema-free

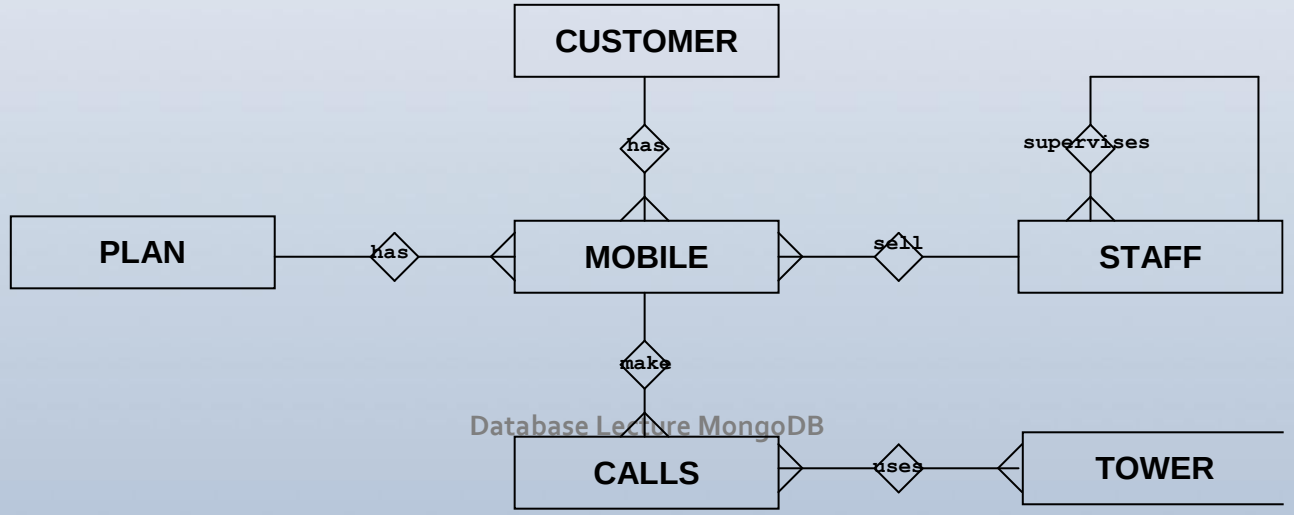
- The purpose of schema in SQL is for meeting the requirements of tables and quirky SQL implementation
- Every "row" in a database "table" is a data structure, much like a "struct" in C, or a "class" in Java. A table is then an array (or list/collection) of such data structures
- Design in mongoDB is basically done the way we design a compound data type binding in BSON (shown earlier which extends JSON)

Mobile Phone Company

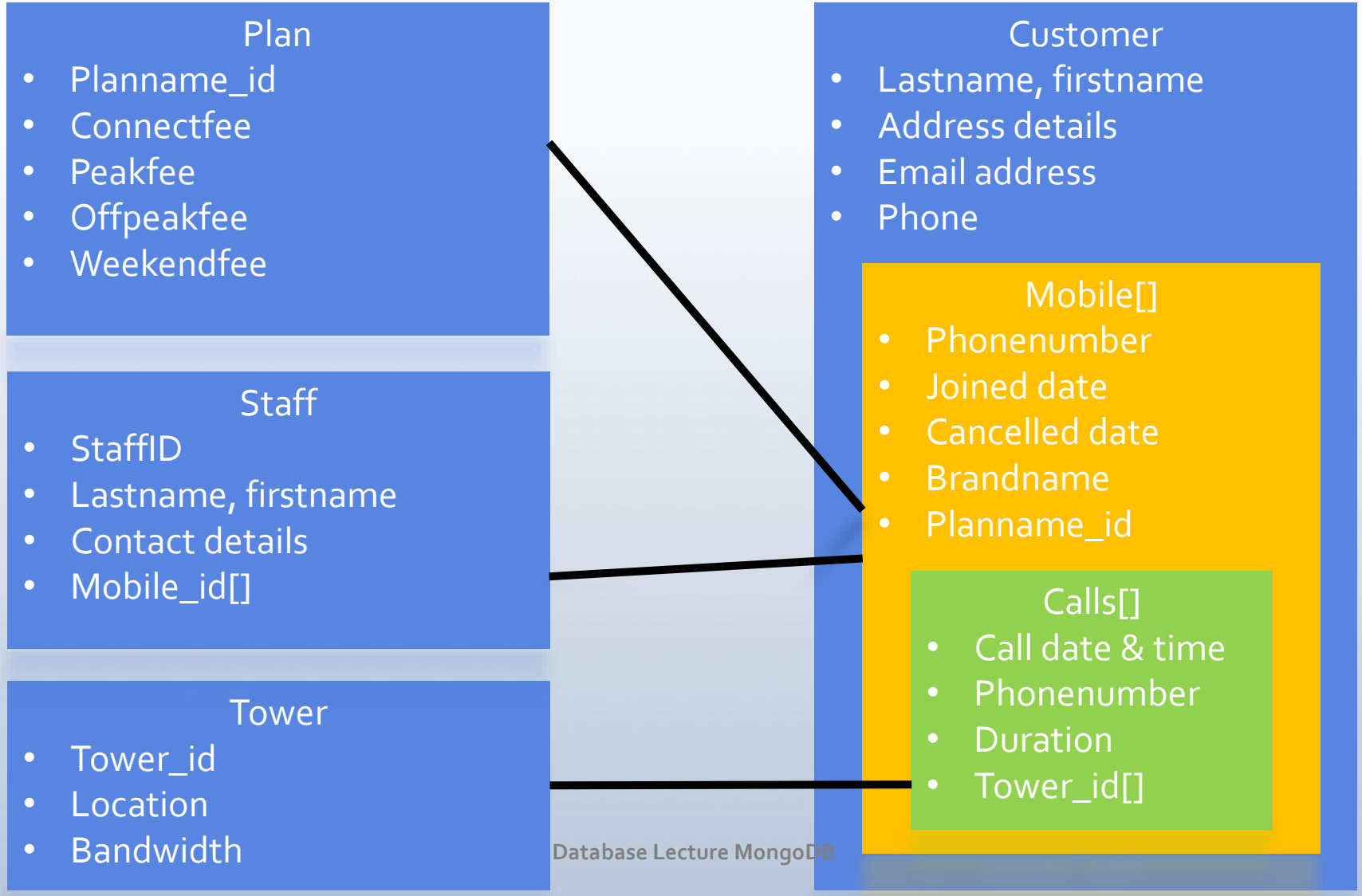
Relational Model

CUSTOMER	CALLS	MOBILE	STAFF	TOWER	CONNECT
CustomerID	CallsID	MobileID	StaffID	TowerID	ConnectID
Surname	MobileID	PhoneNumber	Surname	Location	TowerID
Given	PhoneNumber	BrandName	Given	Bandwidth	CallsID
Dob	CallDate	Joined	Sex	MaxConn	
Sex	CallTime	Cancelled	Joined		
PhoneHome	CallDuration	PlanName	Resigned		
PhoneWork		PhoneColour	Address		PLAN
PhoneFax		CustomerID	Suburb		PlanName
Address		StaffID	Postcode		ConnectFee
Suburb			Phone		PeakFee
State			SupervisorID		OffPeakFee
Postcode			Commission		WeekendFee
			RatePerHour		

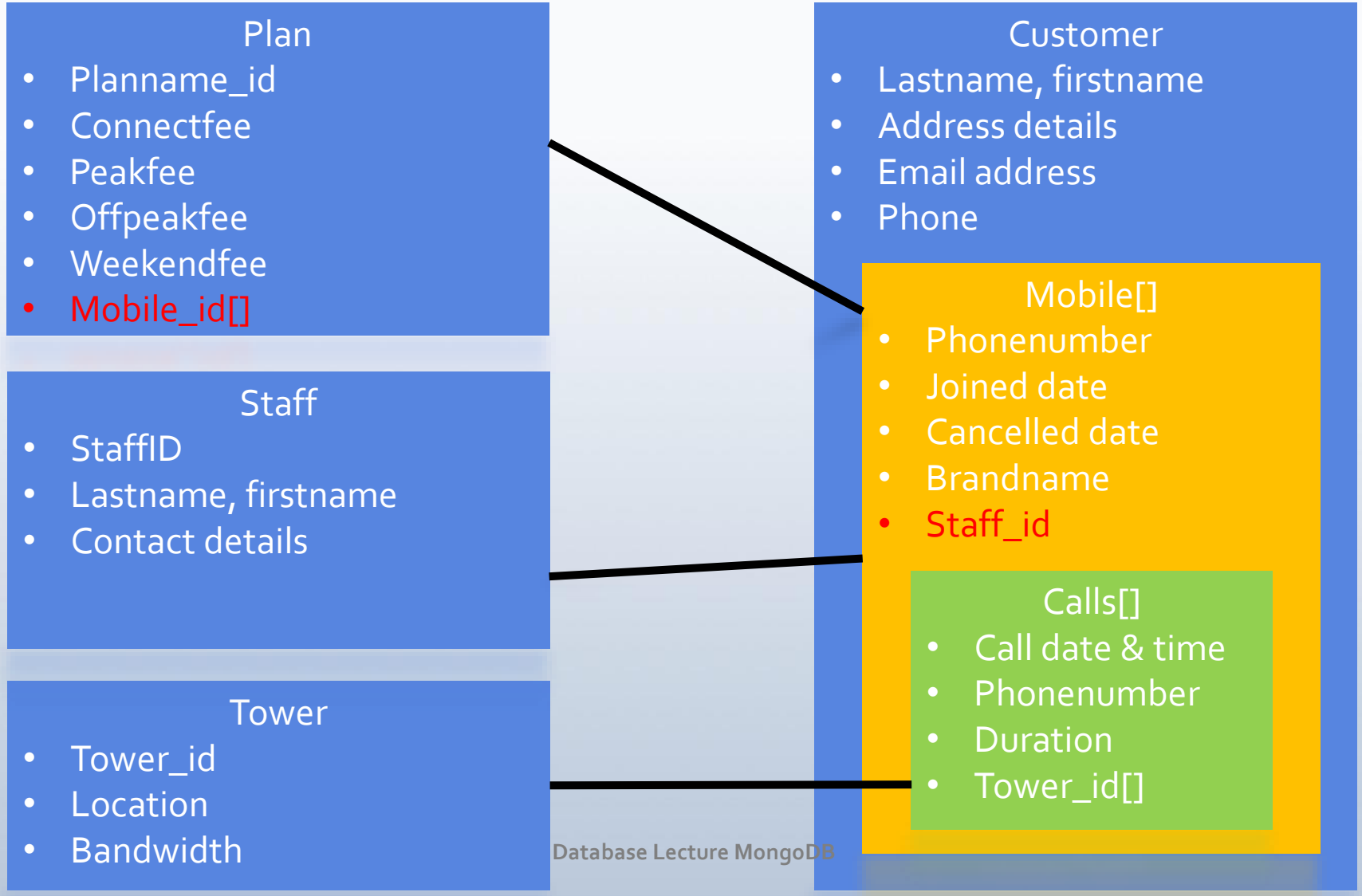
Entity-Relationship Diagram



Embedding & Linking



Linking: Alternative



One to One relationship

Linked	Embedded

zip = {
_id: 35004,
city: “ACMAR”,
loc: [-86, 33],
pop: 6065,
State: “AL”
}
 	
Council_person = {
zip_id = 35004,
name: “John Doe",
address: “123 Fake St.”,
Phone: 123456
}
	zip = {
	_id: 35004,
	city: “ACMAR”
	loc: [-86, 33],
	pop: 6065,
	State: “AL”,
	council_person: {
	name: “John Doe",
	address: “123 Fake St.”,
	Phone: 123456
	}
	}

One to many relationship

Linked	Embedded

Customer = { _id: 1,
lastname: “Flintstone”,
firstname: “Fred”,
address: “8 rock street”,
suburb: “springfield”,
postcode: “3022”,
email: fred@stone.com.au }
 	
Mobile = { _id: 10001,
Phonenumber: “0433222333”,
Joineddate: “10 Sep 2014”,
Brandname: “Nokia”,
Planname_id: “Yes10”,
customer_id: 1,
Calls: [ ] }
Mobile = {_id: 10002,
Phonenumber: “0433222444”,
Joineddate: “12 Sep 2014”,
Brandname: “Apple”,
Planname_id: “Freestyle”,
customer_id: 1,
Calls: [ ] }
	Customer = {
	lastname: “Flintstone”,
	firstname: “Fred”,
	address: “8 rock street”,
	suburb: “springfield”,
	postcode: “3022”,
	email: “fred@stone.com.au”,
	mobile: [{Phonenumber: “0433222333”,
	Joineddate: “10 Sep 2014”,
	Brandname: “Nokia”,
	Planname_id: “Yes10”,
	Calls: [ ]},
	{Phonenumber: “0433222444”,
	Joineddate: “12 Sep 2014”,
	Brandname: “Apple”,
	Planname_id: “Freestyle”,
	Calls: [ ] }]
	}

One to many relationship

Linked	
Customer = { _id: 1, lastname: “Flintstone”, firstname: “Fred”, address: “8 rock street”, suburb: “springfield”, postcode: “3022”, email: fred@stone.com.au, mobile_id: [10001, 10002] }	Mobile = { _id: 10001, Phonenumber: “0433222333”, Joineddate: “10 Sep 2014”, Brandname: “Nokia”, Planname_id: “Yes10”, Calls: [ ] } Mobile = {_id: 10002, Phonenumber: “0433222444”, Joineddate: “12 Sep 2014”, Brandname: “Apple”, Planname_id: “Freestyle”, Calls: [ ] }

Linking vs. Embedding

- Embedding is a bit like pre-joining data
- Document level operations are easy for the server to handle
- Embed when the "many" objects always appear with (viewed in the context of) their parents.
- Linking when you need more flexibility, or worried more about potential for anomalies.
- Linking now possible not just on the many side. A multivalued (array) attribute can act as link on one side.

Many to many relationship

- No longer do you need to create an associative relation (collection, table or file) to handle.
- Can put multivalued (array) attribute of relations in either one of the documents (as a reference/link or embedding in one of the documents)
- Focus on how the data is accessed/queried
- Example: tower_id[] in calls embedded document links to tower collection. A call can have many towers handle connection, a tower can handle many calls. In relational model, it is the norm to create a new relation, i.e. Connect.

Advance Querying

mongoDB queries	SQL queries
db.mobile.distinct("brandname")	SELECT DISTINCT brandname FROM mobile

db.mobile.aggregate([
{
$group: {
_id: null,
count: { $sum: 1 }
}
}
])
	SELECT COUNT(*) AS count
	FROM mobile

Advance Querying

mongoDB queries	SQL queries
db.mobile.aggregate([{$group: { _id: "$customer_id", count: { $sum: 1 } } }, { $match: { count: { $gt: 1 } } }])	SELECT customer_id, count(*) FROM mobile GROUP BY customer_id HAVING count(*) > 1

db.mobile.aggregate([
{$group: { _id: {
cust_id: "$customer_id",
join_date: {
month: { $month: "$ord_date" },
day: { $dayOfMonth: "$ord_date" },
year: { $year: "$ord_date"}
} },
total: { $sum: 1 } } },
{ $match: { total: { $gt: 250 } } }
])
	SELECT customer_id,
	join_date,
	count(*) AS total
	FROM orders
	GROUP BY customer_id, join_date
	HAVING total > 250

Advance Querying

mongoDB queries	SQL queries

db.mobile.aggregate([
{ $unwind: "$calls" },
{
$group: {
_id: "$customer_id",
duration: { $sum: "$calls.duration" }
}
}
])
	SELECT customer_id,
	SUM(duration) as qty
	FROM mobile m,
	calls c
	WHERE m.mobile_id = c.mobile_id
	GROUP BY customer_id

JOIN Query

- Not something that mongoDB does naturally.
- The data schema can be setup in a denormalised way to remove the need for JOINing, i.e. everything is embedded!
- But, it is possible, but a little complex… here is an example:

http://blog.knoldus.com/2014/03/12/easiest-way-to-implement-joins-in-mongodb-2-4/

Summary

- Querying is NOT SQL, each nosql database has its own querying language, there is no standard.
- Data modelling at the conceptual level, ERD can be seen as still a key planning document, BUT…
- Converting it into a low level model (i.e. an implementation model like the relational model), it has been shown that with mongoDB that the concept of embedded document and the use of multi-valued attributes can provide flexibility in the way relationship between the entities are mapped.

Summary

- Mapping relationship in our ERD, for example….
  - One to Many relationship can be mapped by
    1. Placing the foreign key on the many side
    2. Placing a multi-valued foreign key on the one side
    3. Placing the many entity as an embedded document within the one entity
  - Many to Many relationship can be mapped by
    1. Create an associative entity and place the primary keys from both side into it.
    2. Placing a multi-valued foreign key on the either side