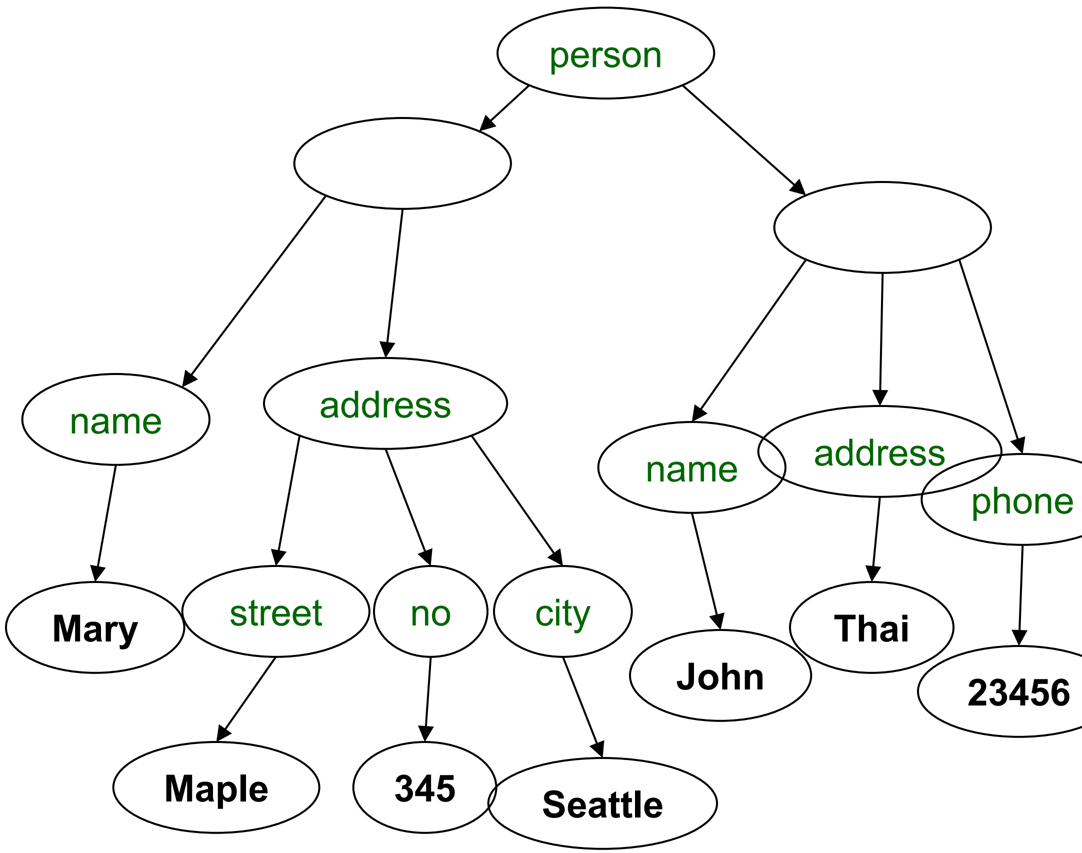
CSE 344

APRIL 18TH – SQL++

JSON SEMANTICS: A TREE !

```
{“person”:
   [ {“name”: “Mary”,
      “address”:
         {“street”:“Maple”,
           “no”:345,
           “city”: “Seattle”}},
     {“name”: “John”,
      “address”: “Thailand”,
      “phone”:2345678}}
    ]
}
```



QUERY LANGUAGES FOR SS DATA

XML: XPath, XQuery (see end of lecture, textbook)

- Supported inside many RDBMS (SQL Server, DB2, Oracle)
- Several standalone XPath/XQuery engines

JSon:

- CouchBase: N1QL, may be replaced by AQL (better designed)
- Asterix: SQL++ (based on SQL)
- MongoDB: has a pattern-based language
- JSONiq http://www.jsoniq.org/

ASTERIXDB AND SQL++

AsterixDB

- No-SQL database system
- Developed at UC Irvine
- Now an Apache project
- Own query language: AsterixQL or AQL, based on XQuery

SQL++

- SQL-like syntax for AsterixQL

ASTERIX DATA MODEL (ADM)

Objects:

- {“Name”: “Alice”, “age”: 40}
- Fields must be distinct:
  {“Name”: “Alice”, “age”: 40, ~~“age”:50~~}

Can’t have repeated fields

Arrays:

- [1, 3, “Fred”, 2, 9]
- Note: can be heterogeneous

Multisets:

- {{1, 3, “Fred”, 2, 9}}

SQL++ OVERVIEW

Data Definition Language (DDL): create a

- Dataverse
- Type
- Dataset
- Index

Data Manipulation Language (DML): select-from-where

DATAVERSE

A Dataverse is a Database

```
CREATE DATAVERSE lec344
CREATE DATAVERSE lec344 IF NOT EXISTS

DROP DATAVERSE lec344
DROP DATAVERSE lec344 IF EXISTS

USE lec344
```

TYPE

Defines the schema of a collection

It lists all *required* fields

Fields followed by ? are *optional*

`CLOSED` **type = no other fields allowed**

`OPEN` **type = other fields allowed**

CLOSED TYPES

```
USE lec344;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   Name : string,
   age: int,
   email: string?
}
```

{"Name": "Alice", "age": 30, "email": "a@alice.com"}

{"Name": "Bob", "age": 40}

-- not OK:
{"Name": "Carol", ”age”: 35, ~~"phone": "123456789"~~}

OPEN TYPES

```
USE lec344;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS OPEN {
   Name : string,
   age: int,
   email: string?
}
```

{"Name": "Alice", "age": 30, "email": "a@alice.com"}

{"Name": "Bob", "age": 40}

-- Now it's OK:
{"Name": "Carol", ”age”: 35, "phone": "123456789"}

TYPES WITH NESTED COLLECTIONS

```
USE lec344;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   Name : string,
   phone: [string]
}
```

```
{"Name": "Carol", "phone": ["1234”]}
{"Name": ”David", "phone": [“2345”, “6789”]}
{"Name": ”Eric",  "phone": []}
```

DATASETS

Dataset = relation

Must have a type

- Can be a trivial `OPEN` type

Must have a key

- Can also be a trivial one

DATASET WITH EXISTING KEY

```
USE lec344;
DROP TYPE PersonType IF EXISTS;
CREATE TYPE PersonType AS CLOSED {
   Name : string,
   email: string?
}
```

```
{"Name": "Alice"}
{"Name": "Bob"}
…
```

```
USE lec344;
DROP DATASET Person IF EXISTS;
CREATE DATASET Person(PersonType) PRIMARY KEY Name;
```

SQL++ OVERVIEW

```
SELECT ... FROM ... WHERE ... [GROUP BY ...]
```

RETRIEVE EVERYTHING

```
{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
     "organization": […],
     …
     …
}
```

```
SELECT x.mondial FROM world x;
```

Answer

```
{"mondial":
      {"country": [ country1, country2, …],
      "continent": […],
      "organization": […],
      …
      …
}
```

RETRIEVE COUNTRIES

```
{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
     "organization": […],
     ...
     ...
}
```

```
SELECT x.mondial.country FROM world x;
```

Answer

```
{"country": [ country1, country2, …],
```

RETRIEVE COUNTRIES, ONE BY ONE

```
{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
     "organization": […],
     ...
     ...
}
```

```
SELECT y as country FROM world x, x.mondial.country y;
```

Answer

country1
country2
...

ESCAPE CHARACTERS

{“mondial”:
  {“country”: [*country1, country2, …*],
  “continent”: […],
  “organization”: […],
  ...
  ...
}

“-car_code” illegal field
Use ` ... `

```
SELECT y.`-car_code` as code , y.name as name
FROM world x, x.mondial.country y order by y.name;
```

Answer

{“code”: “AFG”, “name”: “Afganistan”}
{“code”: “AL”, “name”: “Albania”}
...

NESTED COLLECTIONS

If the value of attribute B is a collection, then we simply iterate over it

```
SELECT x.A, y.C, y.D
FROM mydata as x, x.B as y;
```

x.B is a collection

```
{"A": "a1", "B": [{"C": "c1", "D": "d1"}, {"C": "c2", "D": "d2"}]}
{"A": "a2", "B": [{"C": "c3", "D": "d3"}]}
{"A": "a3", "B": [{"C": "c4", "D": "d4"}, {"C": "c5", "D": "d5"}]}
```

NESTED COLLECTIONS

If the value of attribute B is a collection, then we simply iterate over it

```
SELECT x.A, y.C, y.D
FROM mydata as x, x.B as y;
```

x.B is a collection

{“A”: “a1”, “B”: [{“C”: “c1”, “D”: “d1”}, {“C”: “c2”, “D”: “d2”}]}
{“A”: “a2”, “B”: [{“C”: “c3”, “D”: “d3”}]}
{“A”: “a3”, “B”: [{“C”: “c4”, “D”: “d4”}, {“C”: “c5”, “D”: “d5”}]}

{“A”: “a1”, “C”: “c1”, “D”: “d1”}
{“A”: “a1”, “C”: “c2”, “D”: “d2”}
{“A”: “a2”, “C”: “c3”, “D”: “d3”}
{“A”: “a3”, “C”: “c4”, “D”: “d4”}
{“A”: “a3”, “C”: “c5”, “D”: “d5”}

HETEROGENEOUS COLLECTIONS

```
{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
     "organization": […],
     ...
     ...
}
```

Runtime error

```
SELECT z.name as province_name, u.name as city_name
FROM world x, x.mondial.country y, y.province z, z.city u
WHERE  y.name='Greece';
```

The problem:

```
...
"province": [ ...
   {"name": "Attiki",
    "city" : [ {"name": "Athens"...}, {"name": "Pireus"...}, ..]
     ...},
    {"name": "Ipiros",
    "city" : {"name": "Ioannia"...}
     ...},
```

city is an array

city is an object

HETEROGENEOUS COLLECTIONS

```
{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
     "organization": […],
     ...
     ...
}
```

```
SELECT z.name as province_name, u.name as city_name
FROM world x, x.mondial.country y, y.province z, z.city u
WHERE  y.name='Greece' and is_array(z.city);
```

The problem:

```
...
"province": [ ...
   {"name": "Attiki",
    "city" : [ {"name": "Athens"...}, {"name": "Pireus"...}, ..]
     ...},
    {"name": "Ipiros",
    "city" : {"name": "Ioannia"...}
     ...},
```

Just the arrays

HETEROGENEOUS COLLECTIONS

```
{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
     "organization": […],
     ...
     ...
}
```

Note: get name directly from z

```
SELECT z.name as province_name, z.city.name as city_name
FROM world x, x.mondial.country y, y.province z
WHERE  y.name='Greece' and not is_array(z.city);
```

Just the objects

The problem:

```
...
"province": [ ...
   {"name": "Attiki",
    "city" : [ {"name": "Athens"...}, {"name": "Pireus"...}, ..]
     ...},
    {"name": "Ipiros",
    "city" : {"name": "Ioannia"...}
     ...},
```

HETEROGENEOUS COLLECTIONS

```
{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
     "organization": […],
     ...
     ...
}
```

```
SELECT z.name as province_name, u.name as city_name
FROM world x, x.mondial.country y, y.province z,
         (CASE  WHEN is_array(z.city) THEN z.city
          ELSE  [z.city] END) u
WHERE  y.name='Greece';
```

Get both!

The problem:

```
...
"province": [ ...
   {"name": "Attiki",
    "city" : [ {"name": "Athens"...}, {"name": "Pireus"...}, ..]
     ...},
    {"name": "Ipiros",
    "city" : {"name": "Ioannia"...}
     ...},
```

HETEROGENEOUS COLLECTIONS

```
{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
     "organization": […],
     ...
     ...
}
```

```
SELECT z.name as province_name, u.name as city_name
FROM world x, x.mondial.country y, y.province z,
       (CASE  WHEN z.city is missing THEN []
              WHEN is_array(z.city) THEN z.city
              ELSE  [z.city] END)  u
WHERE  y.name='Greece';
```

The problem:

```
...
"province": [ ...
   {"name": "Attiki",
    "city" : [ {"name": "Athens"...}, {"name": "Pireus"...}, ..]
     ...},
    {"name": "Ipiros",
    "city" : {"name": "Ioannia"...}
     ...},
```

Even better

USEFUL FUNCTIONS

is_array

is_boolean

is_number

is_object

is_string

is_null

is_missing

is_unknown = is_null or is_missing

USEFUL PARADIGMS

Unnesting

Nesting

Group-by / aggregate

Join

Multi-value join

BASIC UNNESTING

An array: [a, b, c]

A nested array: arr = [[a, b], [], [b, c, d]]

`Unnest(arr)` = [a, b, b, c, d]

```
SELECT y
FROM arr x, x y
```

UNNESTING SPECIFIC FIELD

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

UNNESTING SPECIFIC FIELD

A nested collection

coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]

Nested Relational Algebra

Unnest_F(coll) =
[{A:a1, B:b1, G:[{C:c1}]},
 {A:a1, B:b2, G:[{C:c1}]},
 {A:a2, B:b3, G:[]},
 {A:a2, B:b4, G:[]},
 {A:a2, B:b5, G:[]},
 {A:a3, B:b6, G:[{C:c2},{C:c3}]}]

UNNESTING SPECIFIC FIELD

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Nested Relational Algebra

$Unnest_F$(coll) =

```
[{A:a1, B:b1, G:[{C:c1}]},
 {A:a1, B:b2, G:[{C:c1}]},
 {A:a2, B:b3, G:[]},
 {A:a2, B:b4, G:[]},
 {A:a2, B:b5, G:[]},
 {A:a3, B:b6, G:[{C:c2},{C:c3}]}]
```

SQL++

```
SELECT x.A, y.B, x.G
FROM coll x, x.F y
```

Refers to relations defined on the left

UNNESTING SPECIFIC FIELD

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Nested Relational Algebra

Unnest$_F$(coll) =

```
[{A:a1, B:b1, G:[{C:c1}]},
 {A:a1, B:b2, G:[{C:c1}]},
 {A:a2, B:b3, G:[]},
 {A:a2, B:b4, G:[]},
 {A:a2, B:b5, G:[]},
 {A:a3, B:b6, G:[{C:c2},{C:c3}]}]
```

SQL++

```
SELECT x.A, y.B, x.G
FROM coll x, x.F y
```

=

```
SELECT x.A, y.B, x.G
FROM coll x
UNNEST x.F y
```

UNNESTING SPECIFIC FIELD

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Nested Relational Algebra

$Unnest_F$(coll) =

```
[{A:a1, B:b1, G:[{C:c1}]},
 {A:a1, B:b2, G:[{C:c1}]},
 {A:a2, B:b3, G:[]},
 {A:a2, B:b4, G:[]},
 {A:a2, B:b5, G:[]},
 {A:a3, B:b6, G:[{C:c2},{C:c3}]}]
```

$Unnest_G$(coll) =

```
[{A:a1, F:[{B:b1},{B:b2}], C:c1},
 {A:a3, F:[{B:b6}], C:c2},
 {A:a3, F:[{B:b6}], C:c3}]
```

SQL++

```
SELECT x.A, y.B, x.G
FROM coll x, x.F y
```

```
SELECT x.A, x.F, z.C
FROM coll x, x.G z
```

NESTING (LIKE GROUP-BY)

A flat collection

```
coll =
[{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]
```

NESTING (LIKE GROUP-BY)

A flat collection

coll =
[{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]

Nested Relational Algebra

$Nest_A$(coll) =
[{A:a1, GRP:[{B:b1},{B:b2}]}
[{A:a2, GRP:[{B:b2}]}]

$Nest_B$(coll) =
[{B:b1, GRP:[{A:a1},{A:a2}]},
{B:b2, GRP:[{A:a1}]}]

NESTING (LIKE GROUP-BY)

A flat collection

coll =
[{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]

Nested Relational Algebra

$Nest_A$(coll) =
[{A:a1, GRP:[{B:b1},{B:b2}]}
[{A:a2, GRP:[{B:b2}]}]

$Nest_B$(coll) =
[{B:b1, GRP:[{A:a1},{A:a2}]},
{B:b2, GRP:[{A:a1}]}]

```
SELECT DISTINCT x.A,
    (SELECT y.B FROM coll y WHERE x.A = y.A) as GRP
FROM coll x
```

NESTING (LIKE GROUP-BY)

A flat collection

```
coll =
[{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]
```

Nested Relational Algebra

Nest_A(coll) =

```
[{A:a1, GRP:[{B:b1},{B:b2}]}
 [{A:a2, GRP:[{B:b2}]}]
```

Nest_B(coll) =

```
[{B:b1, GRP:[{A:a1},{A:a2}]},
 {B:b2, GRP:[{A:a1}]}]
```

```
SELECT DISTINCT x.A,
    (SELECT y.B FROM coll y WHERE x.A = y.A) as GRP
FROM coll x
```

```
SELECT DISTINCT x.A, g as GRP
FROM coll x
LET g =  (SELECT y.B FROM coll y WHERE x.A = y.A)
```

GROUP-BY / AGGREGATE

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Count the number of elements in the F collection

GROUP-BY / AGGREGATE

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Count the number of elements in the F collection

```
SELECT x.A, COLL_COUNT(x.F) as cnt
FROM coll x
```

GROUP-BY / AGGREGATE

Function	NULL	MISSING	Empty Collection
COLL_COUNT	counted	counted	0
COLL_SUM	returns NULL	returns NULL	returns NULL
COLL_MAX	returns NULL	returns NULL	returns NULL
COLL_MIN	returns NULL	returns NULL	returns NULL
COLL_AVG	returns NULL	returns NULL	returns NULL
ARRAY_COUNT	not counted	not counted	0
ARRAY_SUM	ignores NULL	ignores NULL	returns NULL
ARRAY_MAX	ignores NULL	ignores NULL	returns NULL
ARRAY_MIN	ignores NULL	ignores NULL	returns NULL
ARRAY_AVG	ignores NULL	ignores NULL	returns NULL

JOIN

Two flat collection

```
coll1 = [{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]
coll2 = [{B:b1,C:c1}, {B:b1,C:c2}, {B:b3,C:c3}]
```

```
SELECT x.A, x.B, y.C
FROM coll1 x, coll2 y
WHERE x.B = y.B
```

MULTI-VALUE JOIN

Recall: a many-to-one relation should have one foreign key, from “many” to “one”

Sometimes people represent it in the opposite direction, from “one” to “many”:

- The reference is a string of keys separated by space
- Need to use split(string, separator) to split it into a collection of foreign keys

MULTI-VALUE JOIN

```
river =
[{"name": "Donau”, "-country": "SRB A D H HR SK BG RO MD UA”},
 {"name": "Colorado”, "-country": "MEX USA”},
  ... ]
```

MULTI-VALUE JOIN

```
river =
[{"name": "Donau”, "-country": "SRB A D H HR SK BG RO MD UA”},
 {"name": "Colorado”, "-country": "MEX USA”},
  ... ]
```

String

Separator

split(“MEX USA”, “ “) =
[“MEX”, “USA”]

MULTI-VALUE JOIN

```
river =
[{"name": "Donau", "-country": "SRB A D H HR SK BG RO MD UA"},
 {"name": "Colorado", "-country": "MEX USA"},
  ... ]
```

```
SELECT ...
FROM country x, river y,
        split(y. `-country`, " ") z
WHERE x.`-car_code` = z
```

String

Separator

split("MEX USA", " ") =
["MEX", "USA"]

BEHIND THE SCENES

Query Processing on NFNF data:

Option 1: give up on query plans, use standard java/python-like execution

Option 2: represent the data as a collection of flat tables, convert SQL++ to a standard relational query plan

FLATTENING SQL++ QUERIES

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

FLATTENING SQL++ QUERIES

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Flat Representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	c1
3	c2
3	c3

FLATTENING SQL++ QUERIES

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

SQL++

```
SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = 'a1'
```

Flat Representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	c1
3	c2
3	c3

SQL

```
SELECT x.A, y.B
FROM coll x, F y
WHERE x.id = y.parent and x.A = 'a1'
```

SEMISTRUCTURED DATA MODEL

Several file formats: Json, protobuf, XML

The data model is a tree

They differ in how they handle structure:

- Open or closed
- Ordered or unordered

CONCLUSION

Semistructured data best suited for *data exchange*

For quick, ad-hoc data analysis, use a native query language: SQL++, or AQL, or XQuery

- Modern, advanced query processors like AsterixDB / SQL++ can process semistructured data as efficiently as RDBMS

For long term data analysis: spend the time and effort to normalize it, then store in a RDBMS