Running JavaScript Inside the Database
Data Base Management System (DBMS)

Definition

• A database is an organized collection of data.
• DBMS is a computer software - toolset - that interacts with the user, other applications, and the database itself to capture and analyze data.
• DBMS is only as useful as what you can do with it.
• Everything is about efficiency of computation.
Relational Database

History

Edgar F. Codd proposes a relational model in 1970. Relational databases dominate data management.

- Selection
- Projection
- Cartesian product (cross product, cross join)
- Union
- Set difference (complement, intersection)
Evolution of Computing Infrastructure

History

Google discovers that all of web does not fit in a relational database

1970 - Mainframes
1995 - Clusters of Commodity Hardware
2005 - Commercially Viable
2010 - 2015

Clusterpoint — Running JavaScript Inside the Database
Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, achieving scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the
MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real-world tasks are amenable in this model, as shown given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with
Evolution of Computing Infrastructure

History

1970: Mainframes

1995: Clusters of Commodity Hardware

2005: MapReduce

2010: Key-Value Store

2015: Commercially Viable

Google discovers that all of web does not fit in a relational database

Jeff Dean, Sanjay Ghemawat et al publish papers on MapReduce and BigTable

Clusterpoint — Running JavaScript Inside the Database
Clusterpoint — Running JavaScript Inside the Database
Clusterpoint

NoSQL Database

- Document oriented (JSON/XML/Binary)
- Distributed (sharded + replicated)
- Schema less
- Transactional (ACID)
- Cloud enabled
- v4 introduces distributed computing engine
db.runCommand(
  { 
    mapreduce: "DenormAggCollection",
    query: {
      filter1: { 'in': [ 'A', 'B' ] },
      filter2: 'C',
      filter3: { '$gt': 123 }
    },
    map: function() { emit(
      { d1: this.Dim1, d2: this.Dim2 },
      { msum: this.measure1, recs: 1, mmin: this.measure1, mmax: this.measure2 < 100 ? this.measure2 : 0 }
    );},
    reduce: function(key, vals) {
      var ret = { msum: 0, recs: 0, mmin: 0, mmax: 0 };
      for(var i = 0; i < vals.length; i++) {
        ret.msum += vals[i].msum;
        ret.recs += vals[i].recs;
        if(vals[i].mmin < ret.mmin) ret.mmin = vals[i].mmin;
        if((vals[i].mmax < 100) && (vals[i].mmax > ret.mmax)) ret.mmax = vals[i].mmax;
      }
      return ret;
    },
    finalize: function(key, val) {
      val.mavg = val.msum / val.recs;
      return val;
    },
    out: 'result1',
    verbose: true
  });
);

SELECT
  Dim1, Dim2,
  SUM(Measure1) AS MSum,
  COUNT(*) AS RecordCount,
  AVG(Measure2) AS MAvg,
  MIN(Measure1) AS MMin
FROM DenormAggTable
WHERE (Filter1 IN ('A', 'B'))
AND (Filter2 = 'C')
AND (Filter3 > 123)
GROUP BY Dim1, Dim2
HAVING (MMin > 0)
ORDER BY RecordCount DESC
LIMIT 4, 8

6 Aggregate filtering must be applied to the result set, not in the map/reduce.
7 Ascending: 1; Descending: -1

Grouped dimension columns are pulled out as keys in the map function, reducing the size of the working set.
Measures must be manually aggregated.
Aggregates depending on record counts must wait until finalization.
Measures can use procedural logic.
Filters have an ORM/ActiveRecord-looking style.
New query language

Query language you have never heard of but you are already an expert?!
Technology top 2015 (StackOverflow)

- Javascript: 54.4%
- SQL: 48.0%
- Java: 37.4%
- C#: 31.6%
- PHP: 29.7%
- Python: 23.8%
- C++: 20.6%
- C: 16.4%
- Node.js: 13.3%
- AngularJS: 13.3%
- Ruby: 8.0%
- Objective-C: 7.8%
<table>
<thead>
<tr>
<th>SQL</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>flexible to express queries</td>
<td>hard to express queries</td>
</tr>
<tr>
<td>executes in parallel</td>
<td>difficult to execute in parallel</td>
</tr>
<tr>
<td>static</td>
<td>dynamic</td>
</tr>
<tr>
<td>hard to define expressions</td>
<td>easy to define expressions</td>
</tr>
<tr>
<td>bad with custom routines</td>
<td>great with custom routines</td>
</tr>
</tbody>
</table>
Javascript - V8

Too good to be used only in browsers

• Chrome
• Node.js
• MongoDB
• Google BigQuery UDF
function g () { return 1; }

function f () {
    var ret = 0;
    for (var i = 1; i < 10000000; i++) {
        ret += g ();
    }
    return ret;
}

push rbp
movq rbp,rsp
push rsi
push rdi
subq rsp,0x28

// Save the frame pointer.
// Set the new frame pointer.
// Save the callee's "context object".
// Save the callee's JSFunction object.
// Reserve space for 5 locals.
Javascript - V8

Performance - Problem

Compute the 25,000th prime
For x = 1 to infinity: if x not divisible by any member of an initially empty list of primes, add x to the list until we have 25,000
Javascript - V8

Performance - Contenders

```c++
class Primes {
public:
    int getPrimeCount() const { return prime_count; }
    int getPrime(int i) const { return primes[i]; }
    void addPrime(int i) { primes[prime_count++] = i; }

    bool isDivisible(int i, int by) { return (i % by) == 0; }

    bool isPrimeDivisible(int candidate) {
        for (int i = 1; i < prime_count; ++i) {
            if (isDivisible(candidate, primes[i])) return true;
        }
        return false;
    }

private:
    volatile int prime_count;
    volatile int primes[25000];
};

int main() {
    Primes p;
    int c = 1;
    while (p.getPrimeCount() < 25000) {
        if (p.isPrimeDivisible(c)) {
            p.addPrime(c);
        }
        c++;
    }
    printf("%d\n", p.getPrime(p.getPrimeCount()-1));
}
```

```javascript
function Primes() {
    this.prime_count = 0;
    this.primes = new Array(25000);
    this.getPrimeCount = function() { return this.prime_count; }
    this.getPrime = function(i) { return this.primes[i]; }
    this.addPrime = function(i) {
        this.primes[this.prime_count++] = i;
    }

    this.isPrimeDivisible = function(candidate) {
        for (var i = 1; i <= this.prime_count; ++i) {
            if ((candidate % this.primes[i]) == 0) return true;
        }
        return false;
    }
}

function main() {
    p = new Primes();
    var c = 1;
    while (p.getPrimeCount() < 25000) {
        if (p.isPrimeDivisible(c)) {
            p.addPrime(c);
        }
        c++;
    }
    print(p.getPrime(p.getPrimeCount()-1));
}
main();
```
Javascript - V8

Performance - Results (only 17% slower)

C++

% g++ primes.cc -o primes -O3
% time ./primes
287107

real 0m1.564s
user 0m1.560s
sys 0m0.002s

JavaScript

% time d8 primes-2.js
287107

real 0m1.829s
user 0m1.827s
sys 0m0.010s
Efficiency

- Lazy field binding
- Bind field to index - performance gain
- If no index bind to document
- Concurrent execution
- Narrow down using indices

Index

VS

Accessor

SELECT name.toUpperCase()
FROM db
WHERE name.startsWith("Jo")

Record

Deserialize

SELECT name.toUpperCase()
FROM db
WHERE name.startsWith("Jo")

{ name: "John",
  surname: "Snow"
}
Integration

- C++ Library
- Implements ECMAScript (ECMA-262 5th)
- Accessors - callback that calculates and returns a value when an object property is accessed by a JavaScript
- Interceptors - callback for whenever a script accesses any object property.
JS/SQL

Language structure

• Based on SQL-like structure
• Allows to execute arbitrary JavaScript in any clause of the SELECT or UPDATE statement.
• Native support of JSON and XML data types.
• Joins, nested documents (in v4.1, stay tuned!)
SELECT * FROM product
Insert statement

```
INSERT INTO product JSON VALUE {
  "name": "Schwinn S29 Full Suspension Mountain Bike",
  "image_url": "schwinn_s29.jpeg",
  "description": "...",
  "color": ["black","red"],
  "order_price": 211.16,
  "price": 259.16,
  "packaging": {
    "height": 23,
    "width": 25,
    "depth": 12,
    "weight": 54
  },
  "availability": "In Stock"
}
```
Insert statement

```
INSERT INTO product
(name, image_url, description, color, price, availability)
VALUES
("Schwinn S29 Full Suspension Mountain Bike",
"schwinn_s29.jpeg",
"...",
"black",
259.16,
"In Stock")
```
JS/SQL

Price buckets

<table>
<thead>
<tr>
<th>Condition</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collectible</td>
<td>69</td>
</tr>
<tr>
<td>New</td>
<td>436,499</td>
</tr>
<tr>
<td>Refurbished</td>
<td>96</td>
</tr>
<tr>
<td>Used</td>
<td>4,349</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Under $25</td>
<td>173,356</td>
</tr>
<tr>
<td>$25 to $50</td>
<td>97,659</td>
</tr>
<tr>
<td>$50 to $100</td>
<td>77,298</td>
</tr>
<tr>
<td>$100 to $200</td>
<td>44,941</td>
</tr>
<tr>
<td>$200 &amp; Above</td>
<td>45,587</td>
</tr>
</tbody>
</table>

Discount:
- 10% Off or More (114,200)
- 25% Off or More (73,886)
- 50% Off or More (28,619)
- 70% Off or More (4,818)

Dynacraft 8108-91TJ Girls Hello Kitty Cruiser Bike, Black/Pink/White, 20-Inch
by Dynacraft

$54.77 $139.99
FREE Shipping on orders over $35
Show only Dynacraft items

Clusterpoint — Running JavaScript Inside the Database
JS/SQL

Grouping/Aggregation

```javascript
function PriceBucket(price) {
  var boundaries = [0, 1, 5, 10, 50, 100, 200, 500, 1000];
  for (var i = 1; i < boundaries.length; i++) {
    if (price >= boundaries[i - 1] && price < boundaries[i])
      return boundaries[i - 1].toString() + " to " + boundaries[i].toString();
  }
  return "above " + boundaries[boundaries.length - 1].toString();
}

SELECT PriceBucket(price), COUNT() 
FROM product 
GROUP BY PriceBucket(price);
```

Clusterpoint — Running JavaScript Inside the Database
Aggregating nested documents

```json
{
    "user": "3e9cde95-8077-4386-a35b-fc3b4489dec3",
    "items": [
      {
        "name": "Orange",
        "price": 5,
        "descr": "Special for juice",
        "count": 25
      },
      {
        "name": "Orange",
        "price": 5,
        "descr": "Special for juice",
        "count": 25
      }
    ]
}
```
Aggregating nested documents

```javascript
function sum_items()
{
    var sum = 0;
    for (var i = 0; i < items.length; i++)
        sum += items[i].count * items[i].price;
    return sum;
}

SELECT SUM(sum_items()), AVG(sum_items()), MIN(sum_items()), MAX(sum_items())
FROM baskets
GROUP BY 1
```
JS/SQL

Nested documents (v4.1)

```javascript
{
    name: "Schwinn S29 Full Suspension Mountain Bike",
    price: 259.16,
    inventory: [
        {location: "Warehouse-East", items: 17},
        {location: "Warehouse-West", items: 50}
    ]
};
```
Nested documents (v4.1)

```sql
INSERT INTO product["34A40855"] JSON VALUE {
    name: "Schwinn S29 Full Suspension Mountain Bike",
    price: 259.16
};

INSERT INTO product["34A40855"].inventory JSON VALUE {
    location: "Warehouse-East",
    items: 17
};

INSERT INTO product["34A40855"].inventory JSON VALUE {
    location: "Warehouse-West",
    items: 17
};
```

---

Clusterpoint — Running JavaScript Inside the Database
SELECT price, inventory
FROM product

SELECT location, items, SUPER().name
FROM inventory
WHERE SUPER().price > 30

{  
  name: "Schwinn S29 Full Suspension Mountain Bike",  
  price: 259.16,  
  inventory: [  
    {location: "Warehouse-East", items: 17},  
    {location: "Warehouse-West", items: 50}  
  ]
}
```
INSERT INTO product["34A40855"] JSON VALUE {
    name: "Schwinn S29 Full Suspension Mountain Bike",
    price: 259.16
};

INSERT INTO order JSON VALUE {
    product_key: "34A40855",
    delivery_address: "My Office"
};

SELECT delivery_address, product[product_key].price
FROM order
WHERE product[product_key].price > 20
```
API

REST & more APIs coming soon!

```javascript
$.ajax({
  type: 'POST',
  dataType: 'json',
  data: 'SELECT * FROM DATABASE',
  beforeSend: function (xhr) {
    xhr.setRequestHeader('Authorization', 'Basic ' + btoa('USERNAME:PASSWORD'));
  },
  success: function (data) {
    if (typeof success !== 'undefined') {
      success(data);
    }
  },
  fail: function (data) {
    alert(data.error);
    if (typeof fail !== 'undefined') {
      fail(data);
    }
  }
});
```
Try it!

• Signup for Clusterpoint Cloud account: http://cloud.clusterpoint.com
• Free of charge 10GB of storage
• Be part of community!
SHARE CLUSTERPOINT AND WIN AN IPAD PRO

http://friends.clusterpoint.com
Thank you!