SQLGraph

When ClickHouse marries graph processing

郑天祺 （Amos Bird）
中科院 计算所 网络数据实验室

About Me

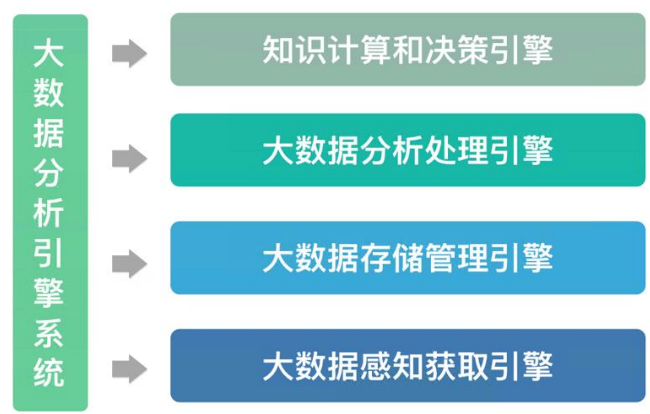
- Active ClickHouse Contributor
  - ~70 valid PRs
  - ~40 Stack Overflow Answers
- Open Source Enthusiast (Hacked 20+ Projects)
  - DB: Impala, Greenplum, Cockroach, Citus, Kudu
  - Misc: emacs, tmux, gdb, fish-shell, tdesktop ...
- SQLGraph Author

https://github.com/amosbird

About My Lab

http://www.ict.ac.cn/jgsz/kyxt/wlzdsys/

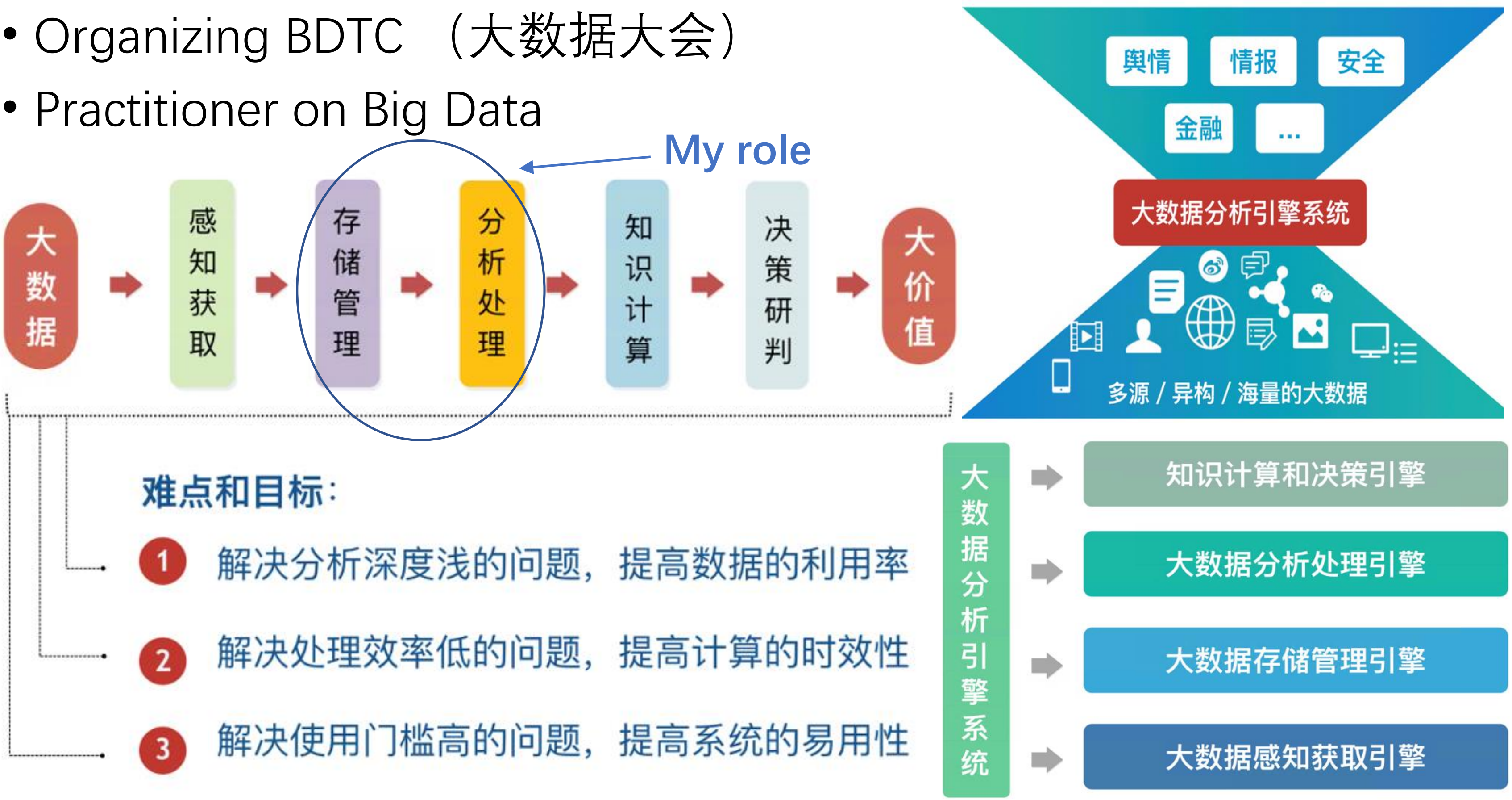
- CCF Task Force on Big Data （大数据专家委员会）
- Organizing BDTC （大数据大会）
- Practitioner on Big Data



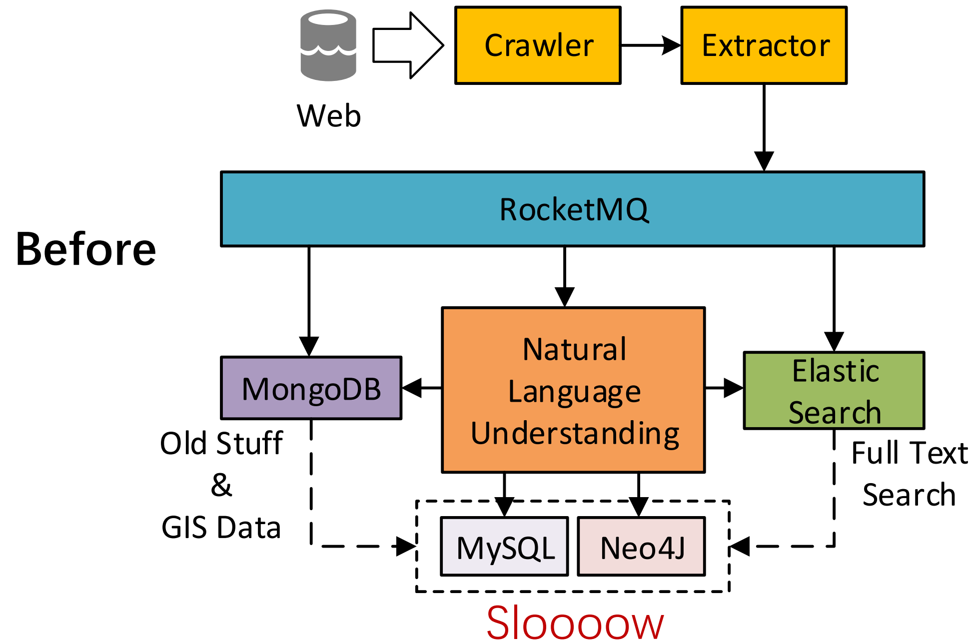
About My Lab

http://www.ict.ac.cn/jgsz/kyxt/wlzdsys/

- CCF Task Force on Big Data （大数据专家委员会）
- Organizing BDTC （大数据大会）
- Practitioner on Big Data



How We Use ClickHouse



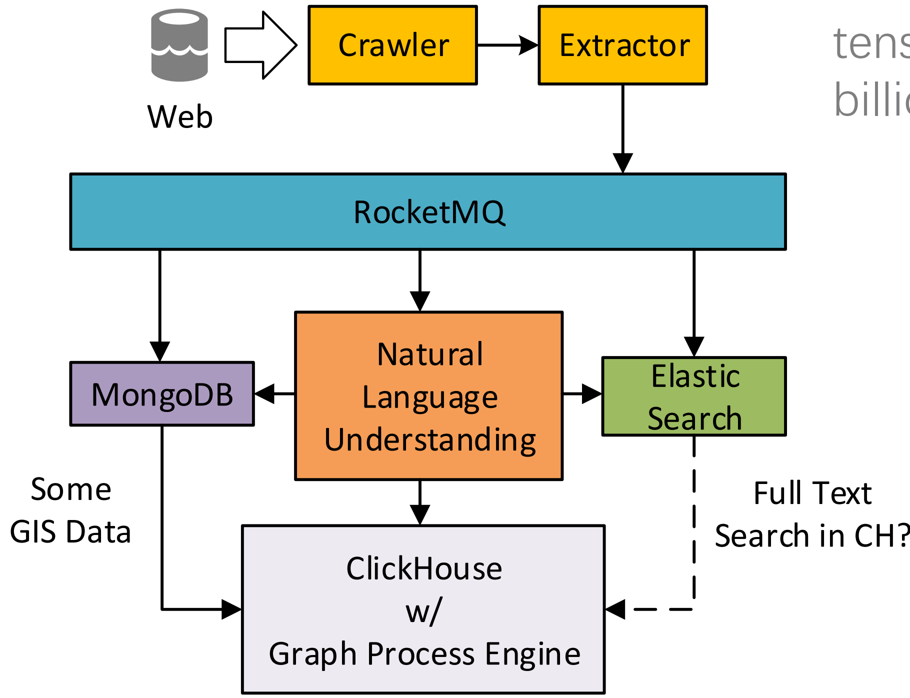
tens of millions docs per day
billions of docs in total

The NLU module outputs well structured, clean, (almost) immutable data.

NLU : Natural Language Understanding

How We Use ClickHouse

After



tens of millions docs per day
billions of docs in total

Swift Graph process engine is a specialized component to run analytical graph algorithms efficiently.

The NLU module outputs well structured, clean, (almost) immutable data.

NLU : Natural Language Understanding

Why Graph Processing Engine

- Finding Important Nodes and Edges
  - PageRank
  - Personalized PageRank
  - Shortest Path

- Community Detection
  - Label Propagation
  - Connection Components
  - Louvain

- Recommendation
  - Collaborative Filtering
  - Random Walk
  - Neural Embeddings

Why Graph Processing Engine in RDBMS

- Hidden graph structures all over the place ☹
  - Graph traversal via joins is both slow and inconvenient
  - Almost impossible to do iterative algorithms (recursion)
  - Cumbersome graph extraction process if acting as a graph data source

- Powerful in data management and processing ☺
  - Full fledged processing (Instead of awk, perl …)
  - Mature concepts of metadata handling
  - Widely adopted and ease of use interface

Mining what exists, aiding what emerges

Why Graph Processing Engine in ClickHouse

- From Architect's Perspective
  - Fast data storage and fast data processing
  - Can handle many data sources
  - Rich user interface

- From Developer's Perspective
  - Nice building blocks for performance critical applications
  - Versatile SQL pipelines for mixed SQL/graph processing
  - Good code quality

Think OLAP but in the graph field.

What to Expect from an OLAP Graph DB

- General Users
  - Easy to use (think as a graph)
  - Easy to understand (view as a graph)
  - Faster than Non-Graph databases or OLTP graph databases
- Graph Experts
  - Extremely fast analysis
  - Interactive graph algorithm designing
  - Ability to do low level optimizations

Be efficient to end users and computer hardware.

System Design Goals

Usability

1. Visualization
2. Graphs Being First-Class Citizens
3. SQL accessibility

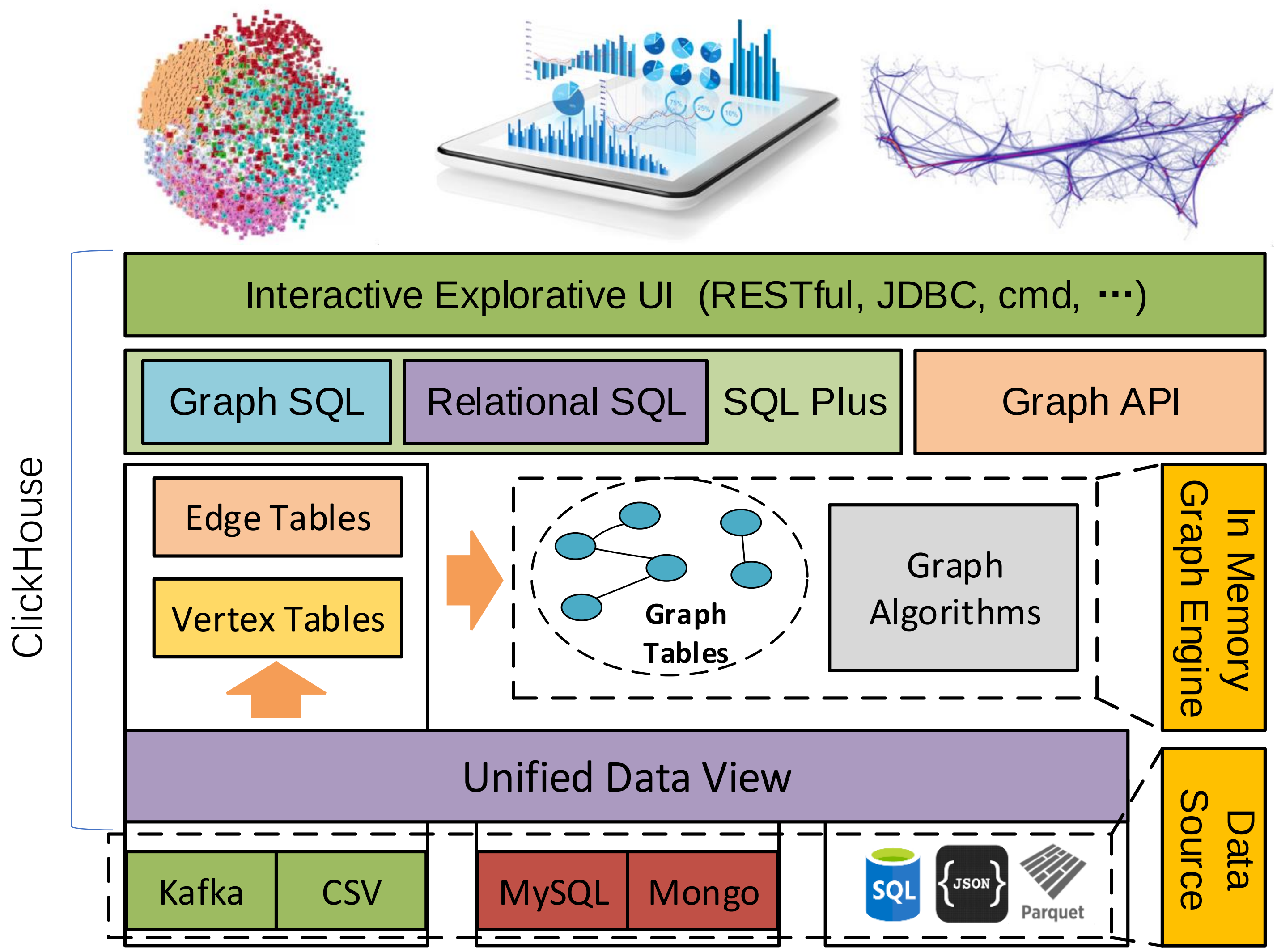
Versatile

1. Graph Algorithm SDK
2. Graph Inspector and Rich Operators
3. Reusable Graph Modeling

Performance

1. Outperform graph compute engines
2. Quick development iterations
3. Fast graph data digestion

SQLGraph


Interactive Explorative UI (RESTful, JDBC, cmd, …)
Graph SQL
Relational SQL
SQL Plus
Graph API
Edge Tables
Vertex Tables
Graph Tables
Graph Algorithms
In Memory Graph Engine
Unified Data View
Kafka
CSV
MySQL
Mongo
SQL
JSON
Parquet
Data Source
ClickHouse

Some Results

Calculate PageRank value per person

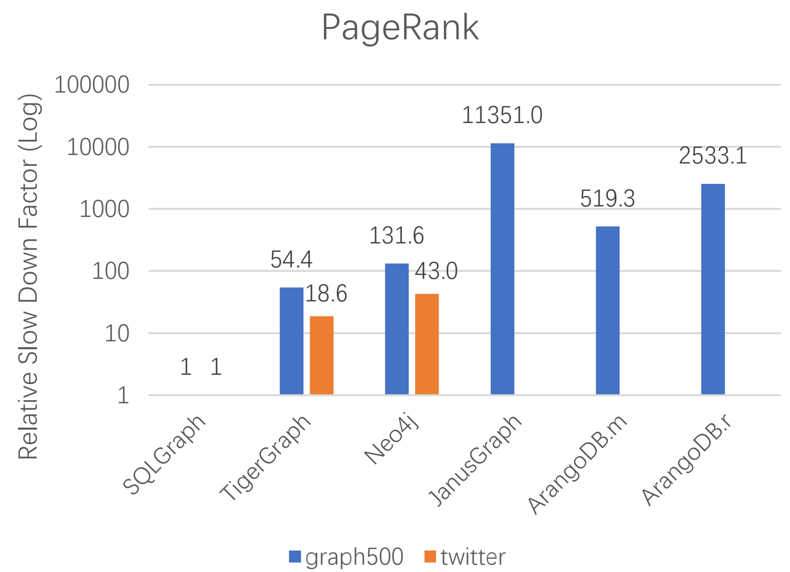
```
SELECT vp(v, 'name') AS name, pagerank
FROM pagerank(wz)
ORDER BY pagerank DESC LIMIT 5
```

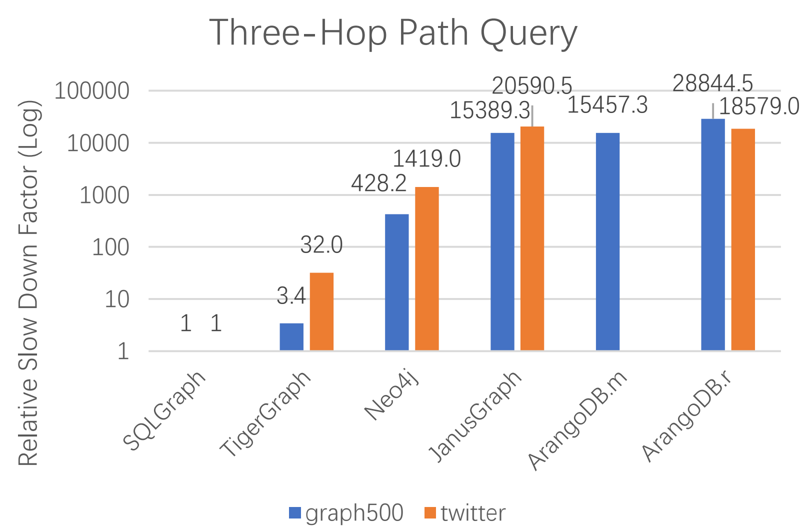
name	pagerank
li	0.06964007
zhao	0.063854694
qian	0.06365149
sun	0.06347877
shen	0.019985389

Find a longest path which ends at ‘shen’

```
SELECT pp(p, 'name', 'w') AS path
FROM edgePath(wz)
WHERE vp(p[-1].2, 'name') = 'shen'
ORDER BY length(path) DESC
LIMIT 1
```

path
zhou--[3]->wang--[2]->chu--[4]->shen





Let’s Dive in

examples and implementations

Graph Query Interface

- Graph Algorithms are table functions
- Graphs are special tables that used as the first argument
- Collection of functions to retrieve graph info

- Example ： Get the top 5 pagerank value from graph “wz”

```
SELECT
    vp(v, 'name’),      --  retrieve the vertex property “name”
    pagerank
FROM pagerank(          --    the graph algorithm
                wz,     --    the graph table
                 5,     --    iterations
              0.85,     --    damping factor
              0.01      --    epsilon
             )
ORDER BY pagerank DESC LIMIT 5
```

Graph Query Output

- Three kinds of outputs : vertices, edges or graph info
- Vertices : PageRank, CC, BC, Radii, etc.
  - Has ‘v : Vertices’ column in the output list
  - Cover all vertices, no duplications
  - Can be inserted back into the graph as a global vertex property

```
SELECT
    id,
    groupArray(vp(v, 'name')) AS names
FROM cc(wz)
GROUP BY id
```

id	names
0	['zhao','qian','sun','li']
4	['zhou','wu','zheng','wang','feng','chen','chu','wei','shen']
14	['han','yang']
12	['jiang']

Graph Query Output

- Three kinds of outputs : vertices, edges or graph info
- Edges : Hop, SSSP, CommonNeighbors, LinkCircle, etc.
  - Has ‘e : Edges’ column in the output list
  - Support function combinators: Out, In, All, Path
  - Can be used to derive subgraphs: create graph as select …
  - Has ‘Graph’ format to support Non-Structural output (for visualization)

```
SELECT pp(p, 'name', 'w')
FROM hopInPath(wz,
(
    SELECT v
    FROM vertex(wz)
    WHERE name = 'zhao'
))
```

```
┌pp(p, 'name', 'w')───────────────────────────────┐
│ zhao<-[1]--li<-[2]--sun<-[3]--qian<-[2]--zhao │
│ zhao<-[1]--li<-[4]--zhou                      │
└───────────────────────────────────────────────┘
```

Graph Query Output

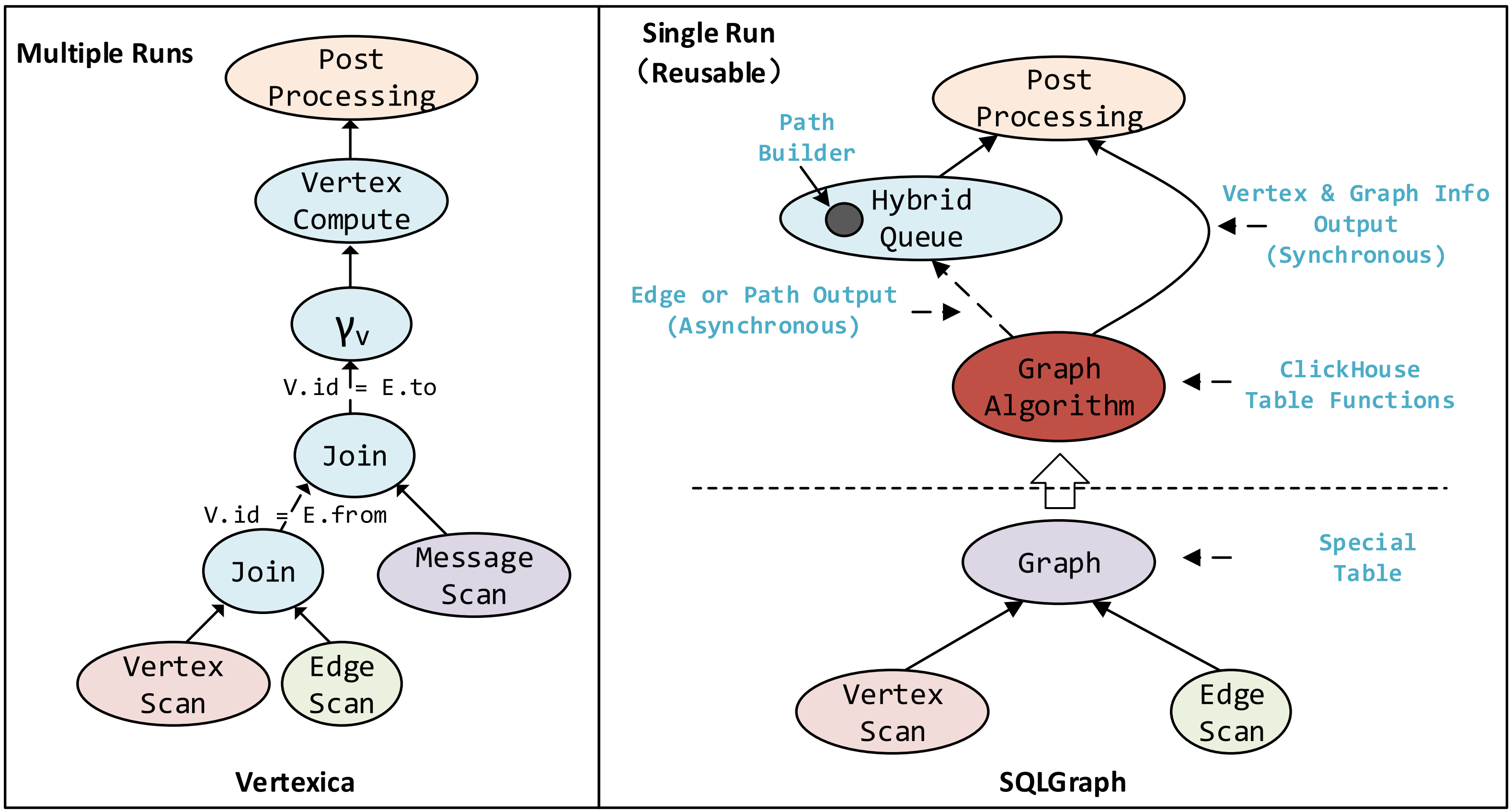
- Three kinds of outputs : vertices, edges or graph info
- Graph info : TriangleCount, MaxClique, ClusteringCoef, etc.
  - Results are graph characters

```
SELECT *
FROM triangle(wz)

┌─count─┐
│     2 │
└───────┘
```

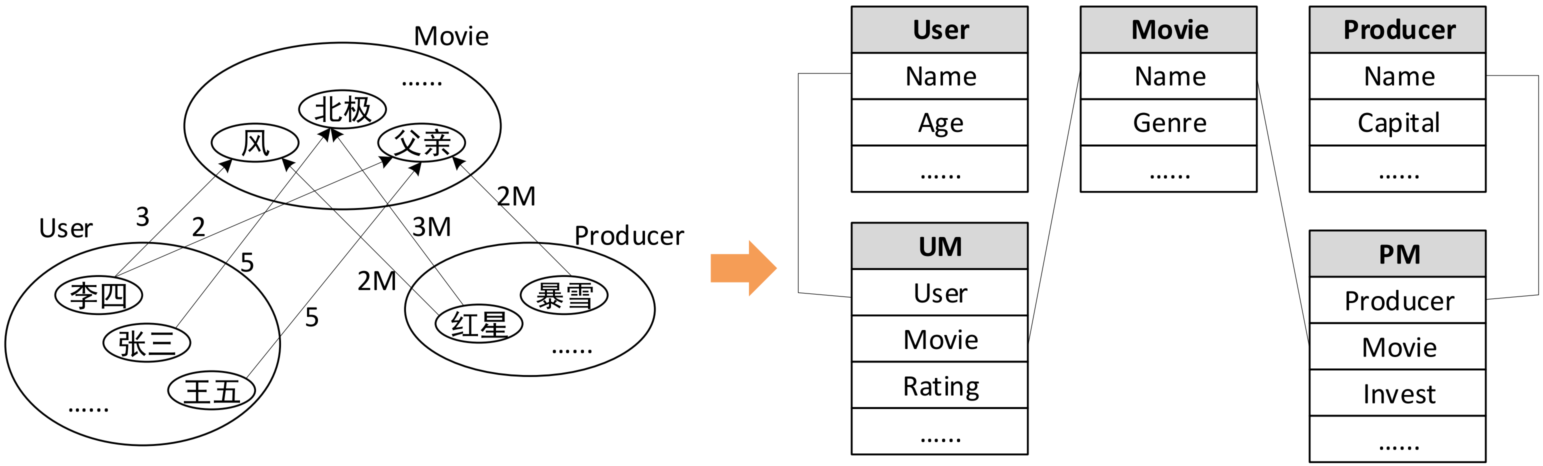
Mainly for graph mining algorithms

Graph Query Output Pipeline (Compared)



How to Get/Build a Graph

What does a graph look like in RDBMS?



How can we make the graph structure efficient?

System \ Algorithm	PageRank (5 iters)	Components
Ligra	6.69s	6.75s
Neo4j	131s	189s

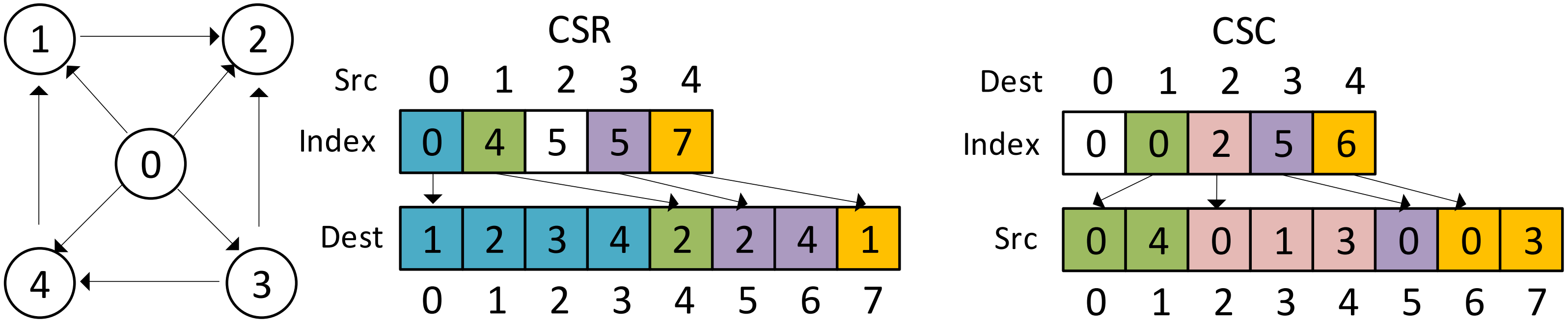
Table 1: Performance Comparison (Twitter-2010)

We need CSR/CSC to be fast!

CSR/C : Compressed Sparse Row/Column

CSR/C Explained

Compressed sparse storage



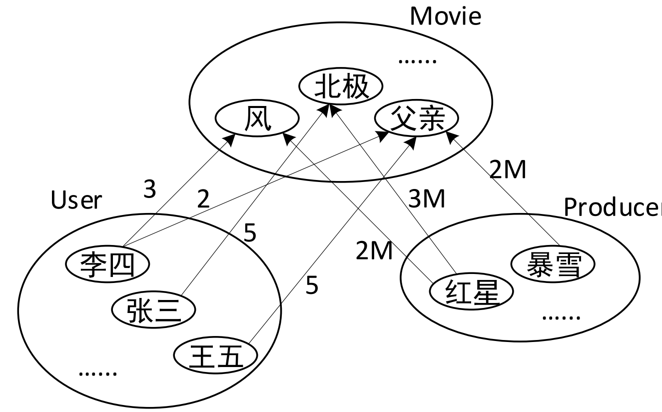
Similar to Arrays in ClickHouse

Vanilla edge list storage

0,1	0,2	0,3	0,4	1,2	3,2	3,4	4,1

Easy to store and modify (normal tables)

From Tables to CSR/C



Define vertex tables to encode key columns

```
CREATE TABLE Producer (Name String key, Capital UInt64) ENGINE V;
CREATE TABLE Movie (Name String key, Genre String) ENGINE V;
CREATE TABLE User (Name String key, Age int) ENGINE V;
```

Key： HashMap
-> [0, P-1]

Property： Columns

Producer
Name: Key
Capital
......

Vertex Table: Producer

红星：0
暴雪：1
E.g.

Key： HashMap
-> [0, M-1]

Property： Columns

Movie
Name: Key
Genre
......

Vertex Table: Movie

风：0
北极：1
父亲：2
E.g.

Key： HashMap
-> [0, N-1]

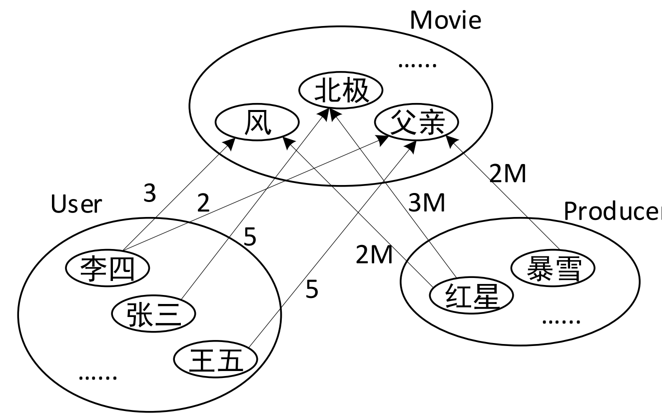
Property： Columns

User
Name: Key
Age
......

Vertex Table: User

张三：0
李四：1
王五：2
E.g.

From Tables to CSR/C



Define edge tables to store relations

```
CREATE TABLE Producer_Movie (Src VS(Producer), Dst VD(Movie), Invest UInt64) ENGINE E;
CREATE TABLE User_Movie (Src VS(User), Dst VD(Movie), Rating int) ENGINE E;
```

src: _VS (Producer)
dst: _VD (Movie)
order by <_VS, _VD>

Property: Columns

Producer_Movie
Producer: VS
Movie: VD
Invest
......

Edge Table: Producer_Movie

红星-风: <0, 0>
红星-北极: <0, 1>
暴雪-父亲: <1, 1>

E.g.

src: _VS (User)
dst: _VD (Movie)
order by <_VS, _VD>

Property: Columns

User_Movie
User: VS
Movie: VD
Rating
......

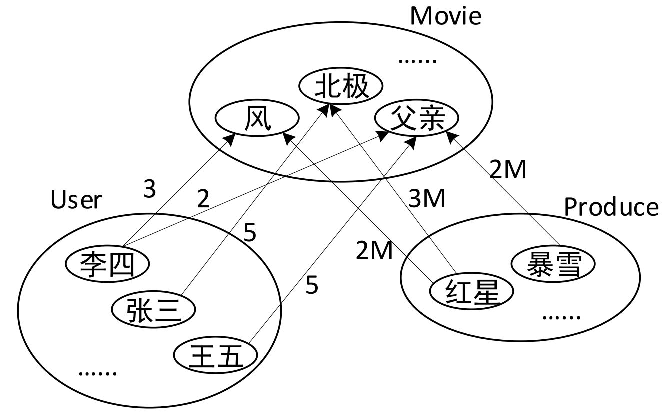
Edge Table: User_Movie

张三-北极: <0, 1>
李四-风: <1, 0>
李四-父亲: <1, 2>
王五-父亲: <2, 2>

E.g.

Storing hidden columns _vs, _vd and use ‘MergeTree order by _vs, _vd’ as the underlying engine

From Tables to CSR/C



Define MetaSQL to specify how to build a graph

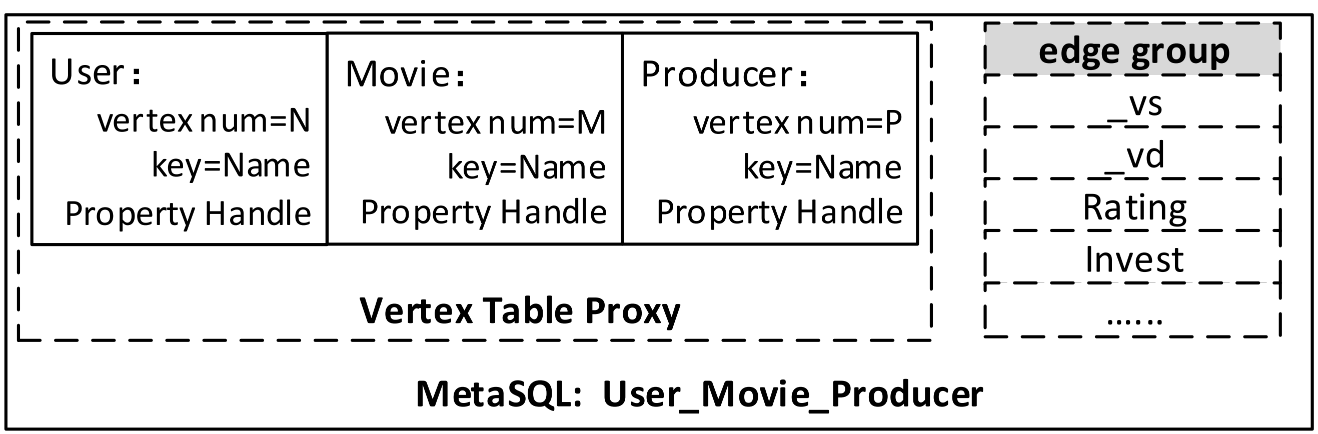
```
CREATE GRAPH User_Movie_Producer AS SELECT * FROM edgeGroup(User_Movie, Producer_Movie)
```

MetaSQL Defines the following workflow:

1. Adjusting IDs of vertices and edges
2. Aggregating edge properties
3. Grouping edges (also assigning Eid)

张三：0
李四：1
王五：2
风：3
北极：4
父亲：5
红星：6
暴雪：7

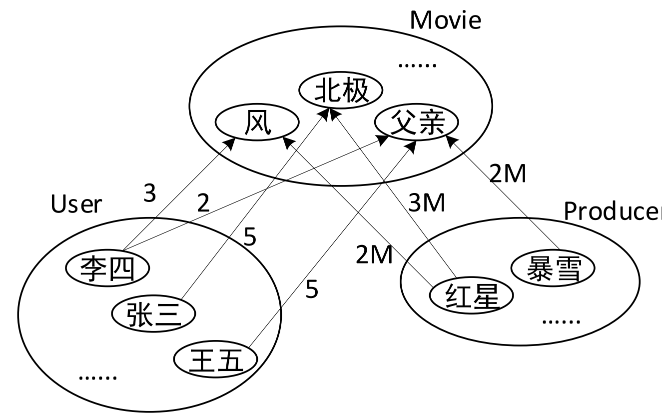
E.g.



	Eid
张三-北极: <0, 4>	1
李四-风: <1, 3>	2
李四-父亲: <1, 5>	3
王五-父亲: <2, 5>	4
红星-风: <6, 3>	5
红星-北极: <6, 4>	6
暴雪-父亲: <7, 5>	7

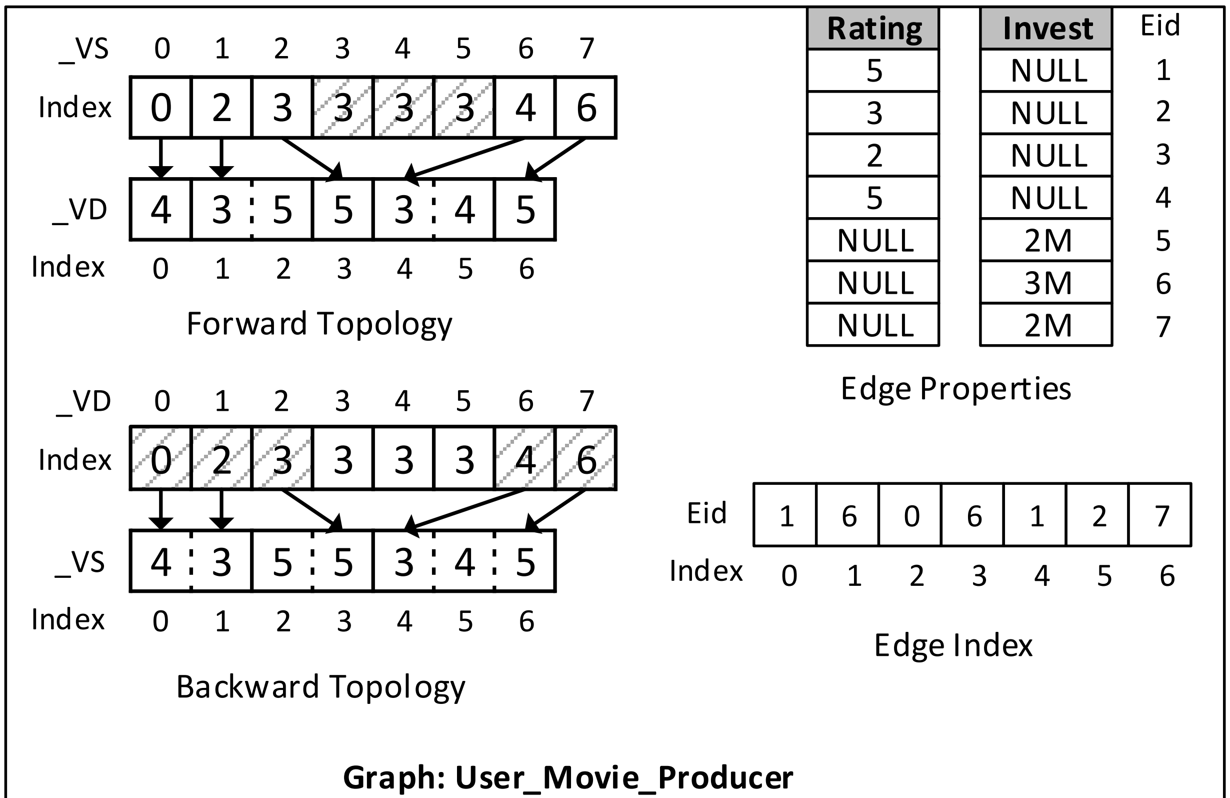
E.g.

From Tables to CSR/C



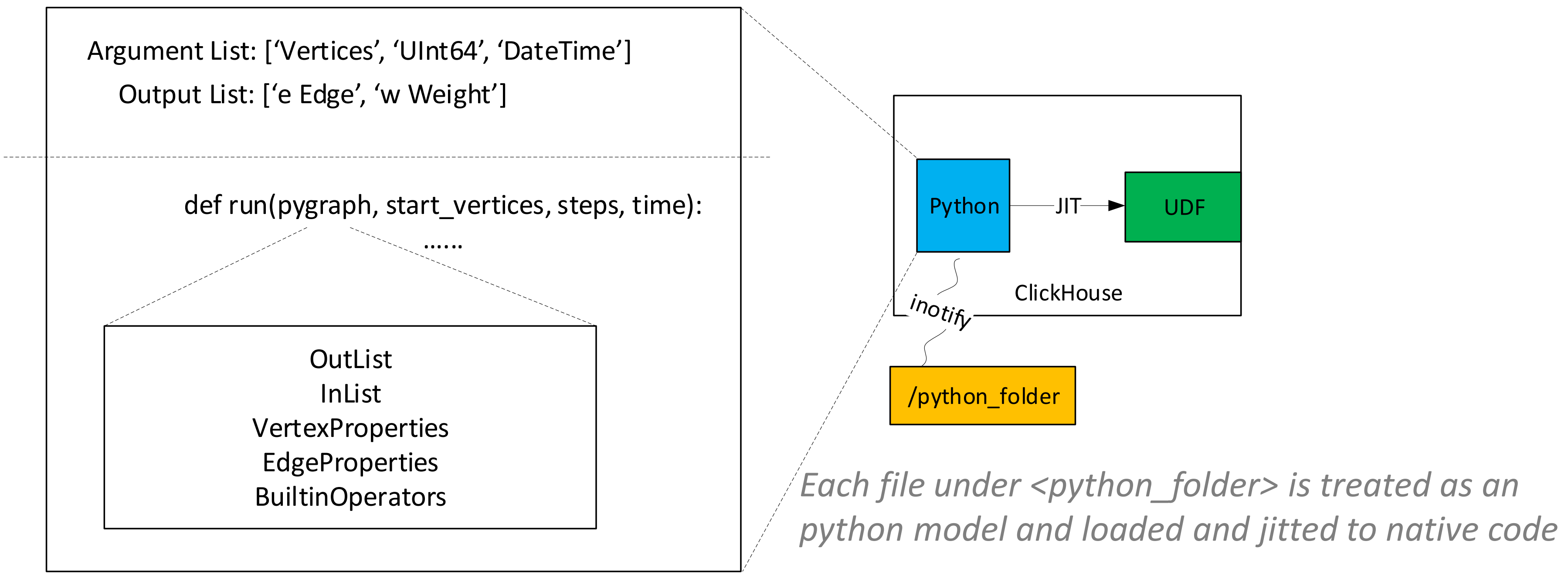
Execute MetaSQL to build a graph

REFRESH User_Movie_Producer [FULL]



How to Write a Graph Algorithm

- Two ways of writing graph algorithms: python or c++
- **Python**: for ad-hoc queries, debugging and performance tuning



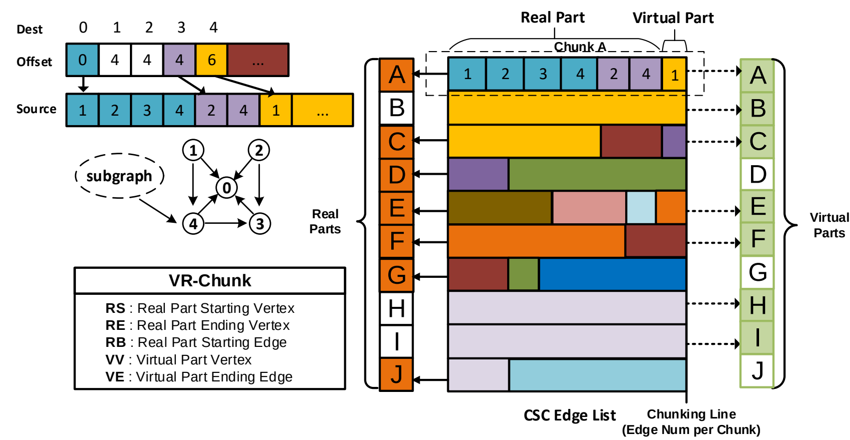
Graph Algorithm NoteBook

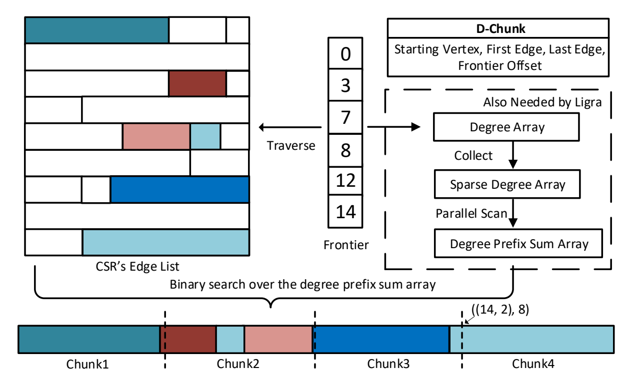
How to Write a Graph Algorithm

- Two ways of writing graph algorithms: python or c++
- **C++**: for long use/well known algorithms, efficiency

```cpp
void PageRankFunction::run(Graph & graph) {
  ... // initialization code
  Algorithm algo(
    [&](UInt32 s, UInt32 d) {// push
      atomicAdd(pr_new[d], pr[s]); },
    [&](UInt32* b, UInt32* e, UInt32 rd, UInt32 d) {// pull
      Float y = 0;
      while (b < e) y += pr[*b++];
      pr_new[d] = y; },
    [&](UInt32 rd, UInt32 d) {// pull reduce
      atomicAdd(pr_new[rd], pr_new[d]); }
  );
  while (!finish) {
    graph.run(algo);
    ... /* other related code */ }}
```

Listing 1. Page Rank Implementation





SilverChunk Framework

Graph Query Helper

- vp, ep, pp for outputting related properties
- v for getting the internal id(s) of a vertex/vertices
- edge, vertex for retrieving edges/vertices of a graph respectively
- show graphs/algos for global inspections
- desc graph/algo for detail info
- graphInfo as a binary interface to access graph info

......

Graph Query Helper (E.g.)

```
DESCRIBE GRAPH wz
```

	name	value or type	comment
Edge properties:			
	region	String	
	time	Date	
	w	Float64	
Vertex group:			
Vertex name:	default.node		
Key name:	id	UInt32	vertex id range : 0 --- 15
Vertex properties:			
	id	UInt32	
	age	UInt32	
	name	String	
Meta info:			
	vertex_num	16	
	edge_num	16	
	symmetric	false	
	load	true	

Graph Query Helper (E.g.)

```
DESCRIBE ALGORITHM pagerank
```

	name	value or type	comment
Arguments:			
		Vertices	vertices : start vertices (default all)
		UInt64	iter : iterations (default 5), 0 for unlimited
		Float64	damping : damping value (default 0.85)
		Float64	epsilon : epsilon value (default 0.001)
Outputs:			
	v	Vertices	
	pagerank	Float32	

Graph Query Helper (E.g.)

```
SHOW ALGOS
```

name	type	comment
bfs	python	
dijkstra	python	
vertexfilter	python	
simplebfs	builtin	
cc	builtin	
bellmanford	builtin	
coregroup	builtin	
linkcircle	builtin	
pagerank	builtin	
bc	builtin	
hop	builtin	
ssp	builtin	
shortcutcc	builtin	
personalrank	builtin	
triangle	builtin	
neighbor	builtin	
closec	builtin	
path	builtin	
commonneighbors	builtin	
radiiest	builtin	
	......	

Limitations

- Single Machine, Main Memory
- **But we have good reasons**
  - Graph size aren't that huge for us (around multiple billions of edges)
    - also commonly seen (referring Sahu, Siddhartha, et al PVLDB 17)
  - Main memory is big, very big (new Mac Pro supports 1.5TB RAM)
  - Very popular in graph computing community (Ligra, Galois, Grazelle, etc.)
  - Nearly impossible to have independent partitions for a graph
    - Commodity networks are too slow, InfiniBand is too expensive
    - Scalability! But at what COST? (McSherry HotOS15)

scalable system	cores	twitter	uk-2007-05
GraphChi [12]	2	3160s	6972s
Stratosphere [8]	16	2250s	-
X-Stream [21]	16	1488s	-
Spark [10]	128	857s	1759s
Giraph [10]	128	596s	1235s
GraphLab [10]	128	249s	833s
GraphX [10]	128	419s	462s
Single thread (SSD)	1	300s	651s
Single thread (RAM)	1	275s	-

20 PageRank iterations

scalable system	cores	twitter	uk-2007-05
Stratosphere [8]	16	950s	-
X-Stream [21]	16	1159s	-
Spark [10]	128	1784s	$\geq$ 8000s
Giraph [10]	128	200s	$\geq$ 8000s
GraphLab [10]	128	242s	714s
GraphX [10]	128	251s	800s
Single thread (SSD)	1	153s	417s

Label Propagation

Future Works

- External Property Partitions
  - Sequential access to properties stored in external storages
- Partial/Incremental Graph Loading
  - Static knowledge database (huge graph) plus loadable domain data
- After the Processor branch Landed
  - Vectorized Graph Queries
  - Streaming and Time Series Processing

Lessons Learned While Extending CH

- Understanding of code
  - semantic tools, utilization of debugger, reading unit tests
  - trial and error
- Isolation of code
  - new modules > new subclasses > Pimpl isolation > in-place patching
  - dlopen for optional features (e.g. python)
- Compilation (linking) is slooooow
  - shared build might help
  - better to build a libclickhouse.so and write main functions per feature
- Contribute while extending

Thank You!

Questions?