# VSFS: A SEARCHABLE DISTRIBUTED FILE SYSTEM

Lei Xu, Ziling Huang, Hong Jiang, Lei Tian, David Swanson

# Introduction

# Introduction



- File systems have been widely used as HPC storage infrastructures
  - Substitutes for databases
  - Better scalability
    - Larger volume
    - Higher parallel I/O performance
  - Flexibility
    - No fixed data schemas
    - Support structured and unstructured data

# Background



- Original file system concepts are aged
  - Were proposed in 1970s
    - Assumption: Single CPU, Small RAM, Small working set, Simple computing model, etc.
  - The assumption does not hold true now
    - Multicore processor
    - Large amount of RAM
    - Large working set
    - Complex computing model

# Big Data Characteristics



- Velocity and Variety
  - Database is insufficient
    - Fixed schema and low throughput
    - Not suitable for scientific dataset
  - Large-scale distributed file systems are the standard solutions today (Hadoop, Ceph, Lustre, Panasas)
- Volume → Management Challenge
  - Difficult to efficiently manage and organize enormous number (e.g., $10^9$) of files for various applications with different access patterns.

# File System Namespace



- File system namespace becomes complex and inefficient for managing large datasets
  - Root cause: file path is the only identity of data
    - Must be *descriptive*
      - Difficult to be distinguishable for billions of files
    - Difficult to locate target file from billions of files
  - Hierarchical namespace does not work well with a huge amount of files.

# File Search – Data Filtering

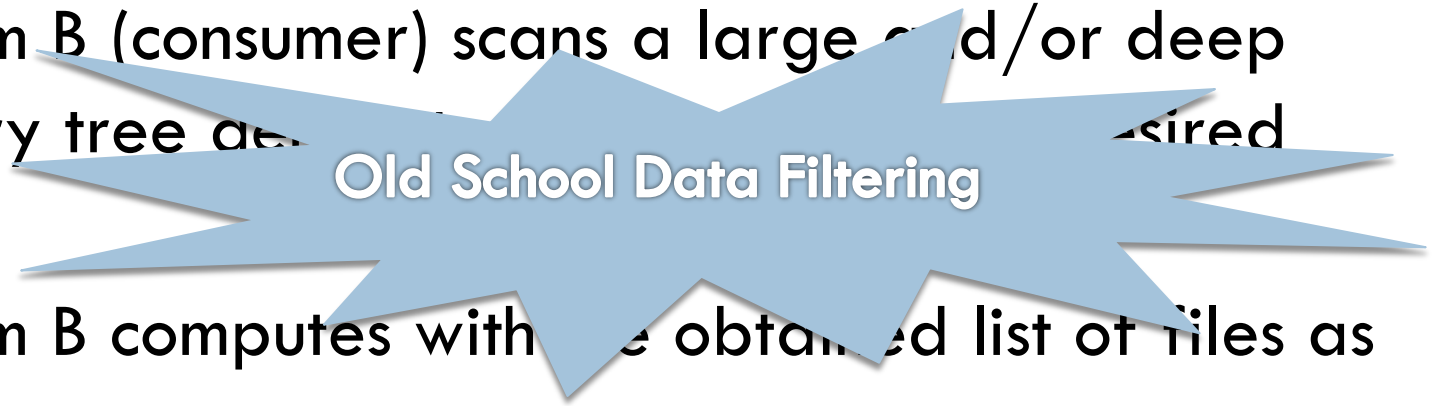


- Addressing data management dilemma
  - Locate (search) files by “attributes” instead of “path”
  - Support high variety
  - Support large volume
  - Speed up big data computing
    - Enable new computing flow

# Today's Computing Flow



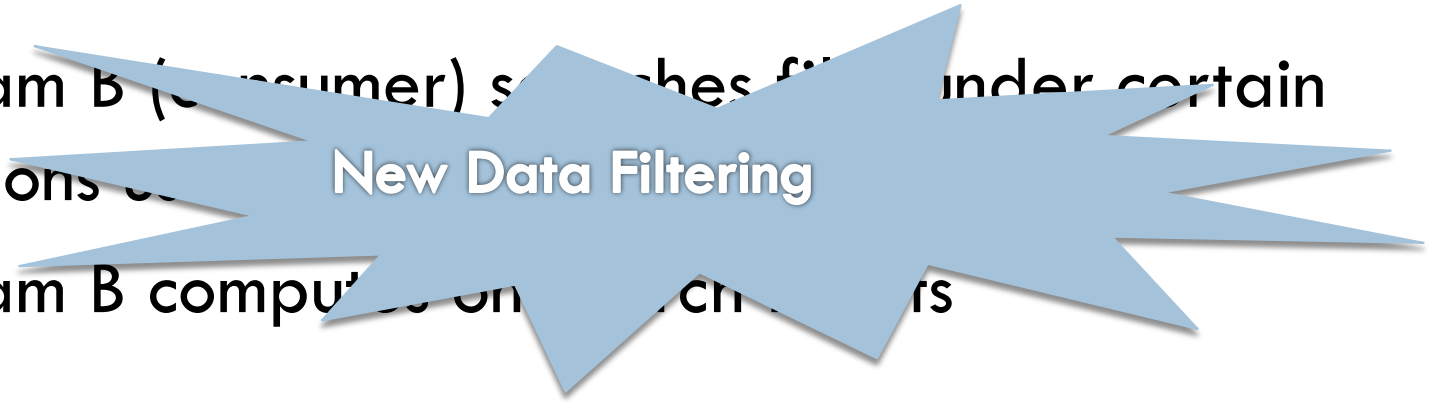
- Program A (producer) writes data into files, with a limited number of attributes embedded into file paths
- Program B (consumer) scans a large and/or deep directory tree and ... desired files
- Program B computes with the obtained list of files as input



# New Computing Flow with Search



- Program A (producer) generates and tags (indexes) files
- Program B (consumer) searches files under certain conditions
- Program B computes on search results




- More flexible (e.g., search attributes rather than file paths)
- More efficient (do not require brute-forced directory scanning).

# Design

# VSFS: A Searchable Distributed File System

- Defines a new file system form
  - Deeply integrates a file-search service
  - Searchable File System
    - File search as first-class API
      - Retrieve files using file-search queries
    - Build filesystem namespace around file-search API

# VSFS: A Searchable Distributed File System (Cont'd)

- Defines a new file system form (Cont'd)
  - Enables existing applications to use file system like using a database!
    - But no data model / code changes required!
    - A new way to interact with file system
      - Enables a new computing model

# Key Points

- Closely couples file search with computing
  - Use file search to assist computing to reduce the input data scale, thus speeding up computing
- A New File Query Language
  - Compatible with existing file system namespace
- Real-Time Indexing
  - Guarantee the consistency of file-search results
- Distributed Architecture

# NFQL



- NFQL: Namespace-based File Query Language
  - Use dynamic directories to represent queries
    - VSFS fills search results in a dynamic directory
    - Thus, scanning this dynamic directory → obtaining file-search results
  - POSIX-compatible
  - Existing applications can use "readdir()" to search, e.g.,
    - `ls /path/data/?attr1>100/`

# NFQL Definition



⟨*query*⟩ := ⟨*prefix*⟩ '/?' ⟨*expression*⟩
[⟨*topk*⟩] ⟨*expression*⟩ := ['('] ⟨*expression*⟩
[')']
| ⟨*expression*⟩ {('&' | '|' ) ⟨*expression*⟩}
| ⟨*range query*⟩ | ⟨*point query*⟩ | ⟨*multi dimensional query*⟩ ⟨*range query*⟩ := ⟨*index*⟩
('>' | '>=' | '<' | '<=') ⟨*value*⟩
⟨*point query*⟩ := ⟨*index*⟩ '=' ⟨*value*⟩
⟨*multi dimensional query*⟩ :=
⟨*index*⟩'['⟨*num*⟩']' ('>' | '>=' | '<'
| '<=') ⟨*value*⟩
⟨*topk*⟩ := '#' ⟨*num*⟩ ['+'|'-']

Example: "/foo/bar/?drug-A:energy> 10.5&weight< 16/"

# Real-Time Indexing



- To support file search, VSFS integrates real-time & "versatile" indexing support
  - Capable of indexing data in real-time
    - Guarantees the consistency between file-search results and file contents.
  - Provide flexibility for indexing data with arbitrary attributes

# Versatile Index



- A file-index is a **versatile** key-value structure defined on a directory, defined as a 4-parameter tuple (root, name, type, key)
  - Root: the directory covered by this index
  - Name: an arbitrary name to identify the index
  - Type: the data structure of index (e.g., b-tree or hash)
  - Key: the numeric type or string type of the key (e.g., `int`)

# RAM-based Index Cluster



- To enable real-time indexing
  - Use in-ram index cluster
    - Keep all file-indices in RAM
      - Periodically flushed to persistent storage
    - Use a consistent hashing ring to scale a single index to multiple nodes for large RAM space.

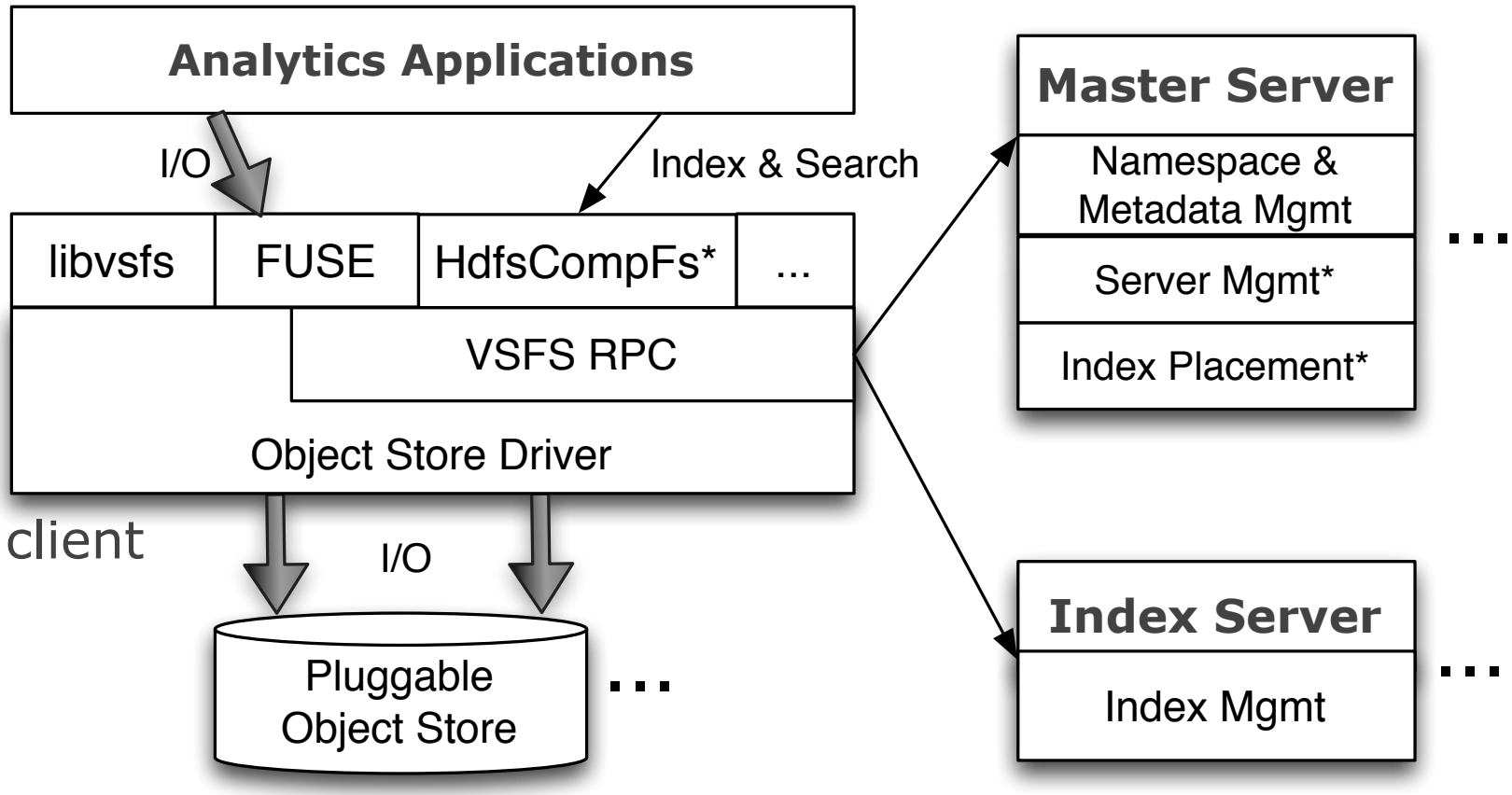
# Distributed Architecture



- Master Server
  - Metadata and namespace management
- Index Server
  - In-memory cluster for file indices
  - Periodlically flushed to persistent storage
- Pluggable Object Store
  - Used for all persistent data
- Client: A library and A FUSE-based file system
  - Dynamic creation of directories for file-search requests

# VSFS Stack




Analytics Applications
I/O
Index & Search
libvsfs
FUSE
HdfsCompFs*
...
VSFS RPC
Object Store Driver
client
I/O
Pluggable
Object Store
...
Master Server
Namespace &
Metadata Mgmt
Server Mgmt*
Index Placement*
...
Index Server
Index Mgmt
...

# Evaluation

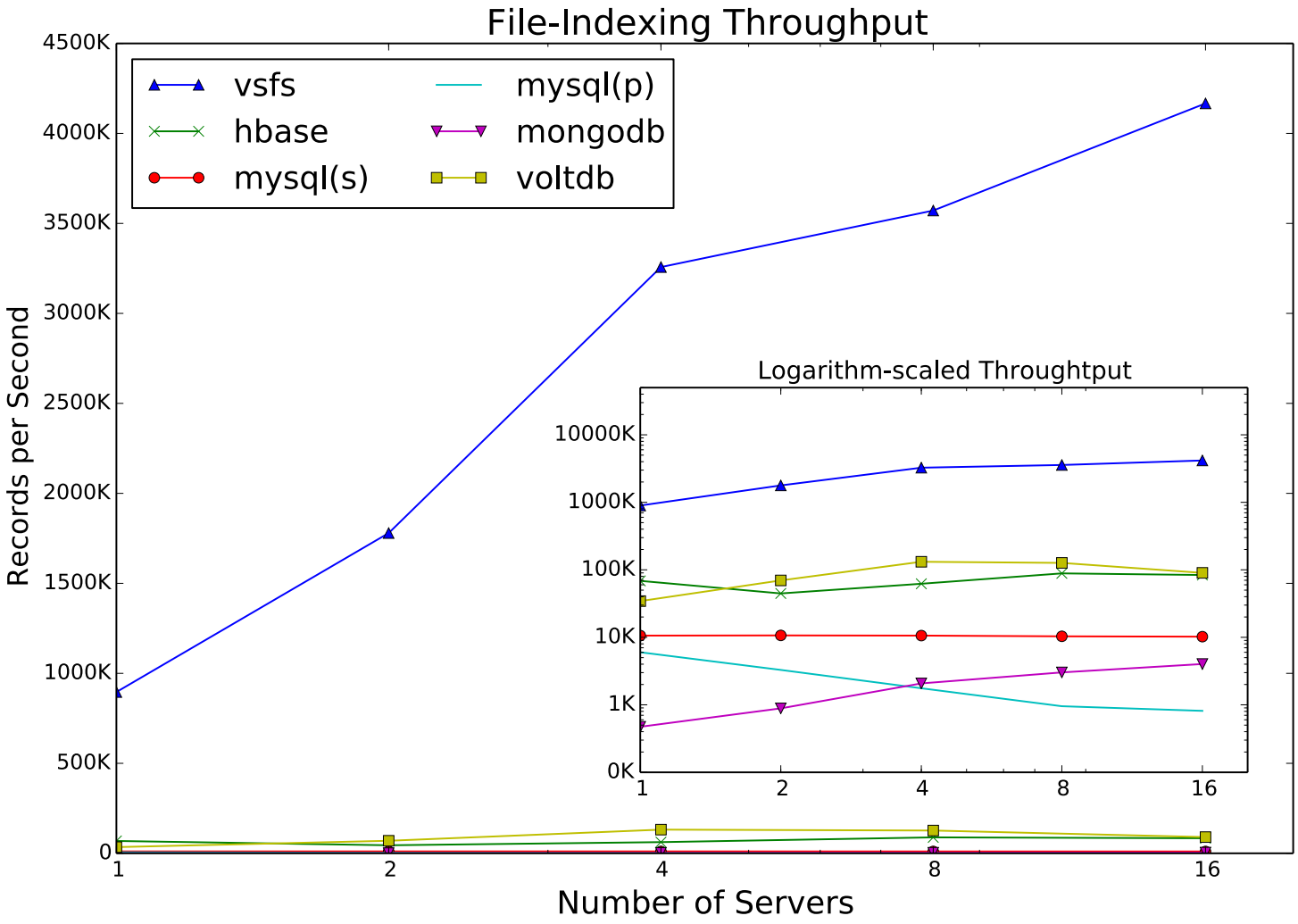
# Evaluation



- Run on a 20-node cluster, 1~16 as index servers
- Compared with SQL (MySQL), NoSQL (MongoDB) and NewSQL (VoltDB, an in-memory SQL)
  - Compare indexing performance
- Directly run existing applications on VSFS (FUSE)
  - Use Lustre as object storage
  - Demonstrate transparent speed up of existing applications (Hive)

# Evaluation (Indexing)





# Evaluation (Hive)



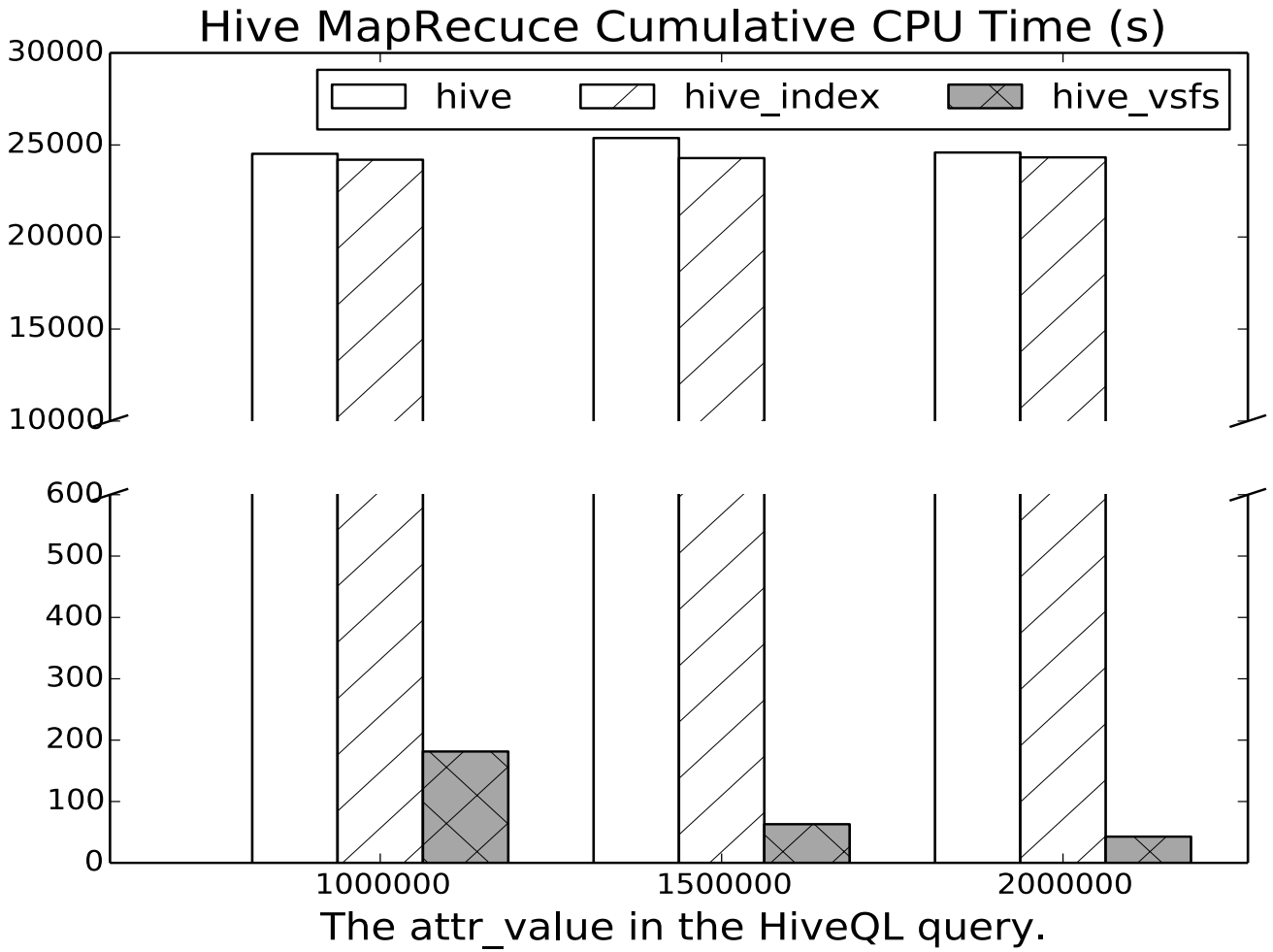
- Most interesting part of this work is VSFS's capability of
  - Transparently integrating w/ existing applications w/o code modification
- We use Hive, a SQL engine on top of Hadoop, as an example.
  - Its code base is too complex to modify!
  - As most real-world applications are!
- Run three modes, all are on the 20-node cluster
  - Machine learning dataset [TrionSort]
  - 3 computing models: Hive, Hive_index and Hive_vsfs

# Query

- HiveQL query to answer:
  - "*find the minute in which the TrionSort cluster contains the highest number of the high-latency events caused by an interesting feature*"
  - 
```
SELECT minute, count(minute) AS mincount FROM
(SELECT round(time / 60) AS minute FROM trionsort
WHERE attr_name = 'Writer_5_runtime' and attr_value
> 2000000) t2 GROUP BY minute ORDER BY mincount DESC
LIMIT 1;
```

# Hive Execution Time





# Hive

- Searching as a common facility in file system has shown its performance advantages.
- Encourages the applications to take advantage of the search functionality.
  - Usually it only incurs minimal effort.

# Conclusion

- VSFS demonstrates that searching a as file system facility can significantly improve existing application performance.
  - Higher abstraction of manipulating data.
- NFQL offers backward-compatibility to the existing applications.
- RAM-based index scheme and distributed architecture

# Q & A