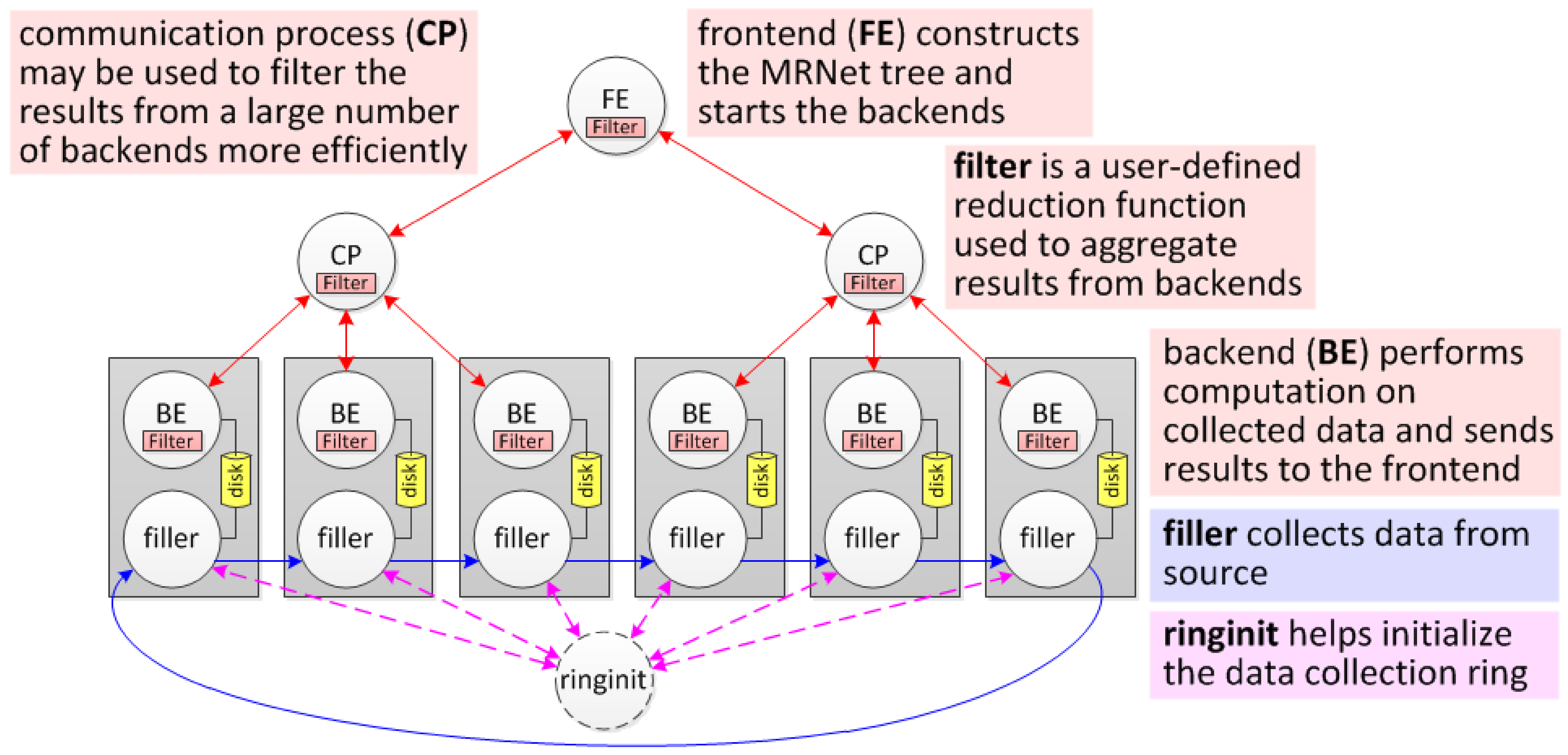
WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Bonsai: A Distributed Data Collection and Storage System for Recent and Historical Data Processing

Kritphong Mongkhonvanit and Kai Da Zhao

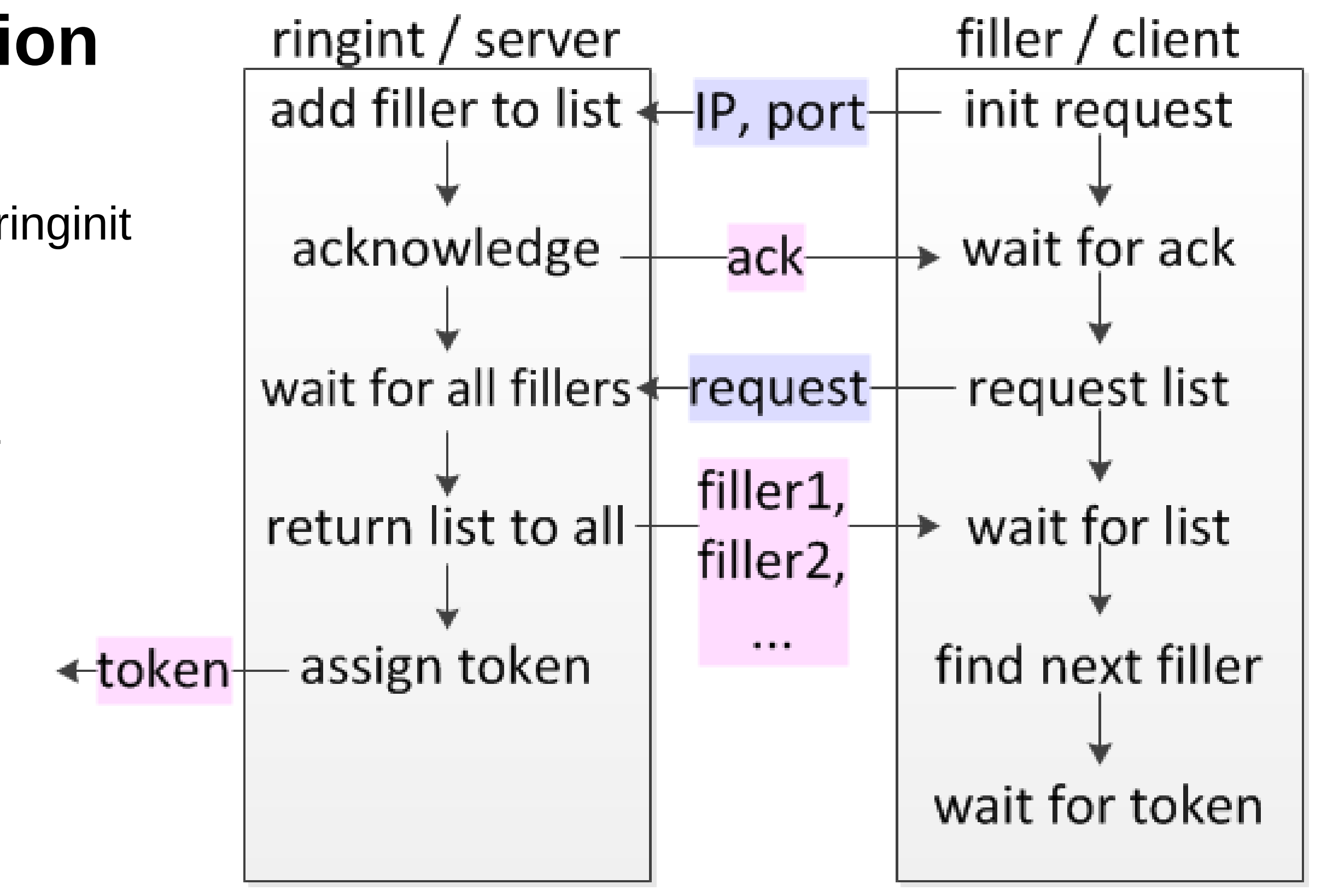
Architecture


communication process (**CP**) may be used to filter the results from a large number of backends more efficiently
frontend (**FE**) constructs the MRNet tree and starts the backends
filter is a user-defined reduction function used to aggregate results from backends
backend (**BE**) performs computation on collected data and sends results to the frontend
filler collects data from source
ringinit helps initialize the data collection ring
FE
Filter
CP
Filter
BE
Filter
disk
filler
ringinit

Data collection ring, shown on the bottom half of the figure, contains ringinit and filler. Data processing tree, shown on the top half of the figure, is MRNet. MRNet consists of a frontend, communication processes, backends, and filters.

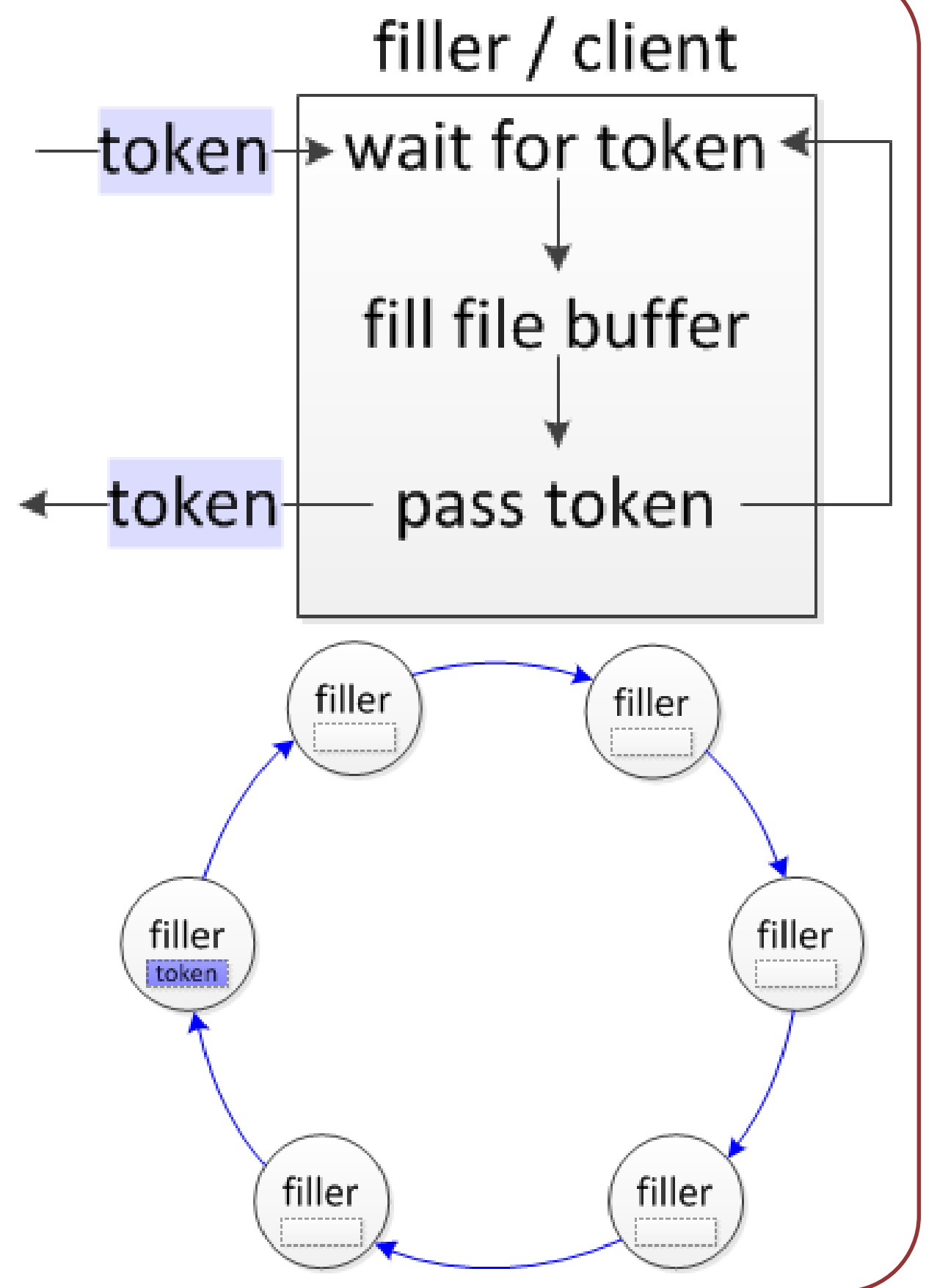
Data Collection Initialization

- filler
  - Announce IP and port number to ringinit
  - Poll ringinit for list of fillers
  - Find next filler from list
- ringinit
  - Record filler's IP and port number
  - Return full list of fillers
  - Send initial token



Data Collection

- filler
  - Wait for token and start collecting data when it is received
  - Once the buffer file is filled, pass the token to the next filler
- Token
  - Synchronization logic to inform filler to start collecting
- Buffer file
  - File on disk to contain collected data
  - Data will be overwritten on record-by-record basis

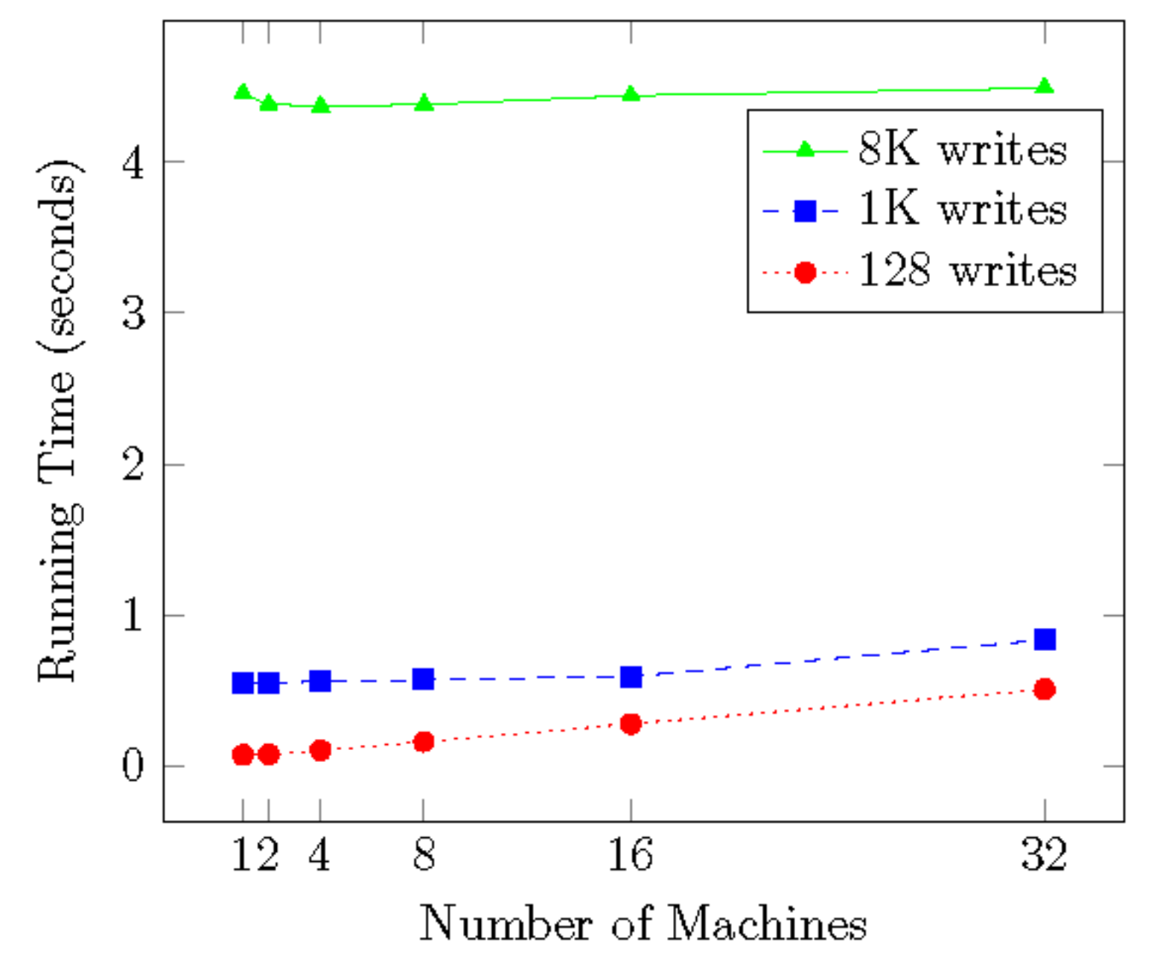


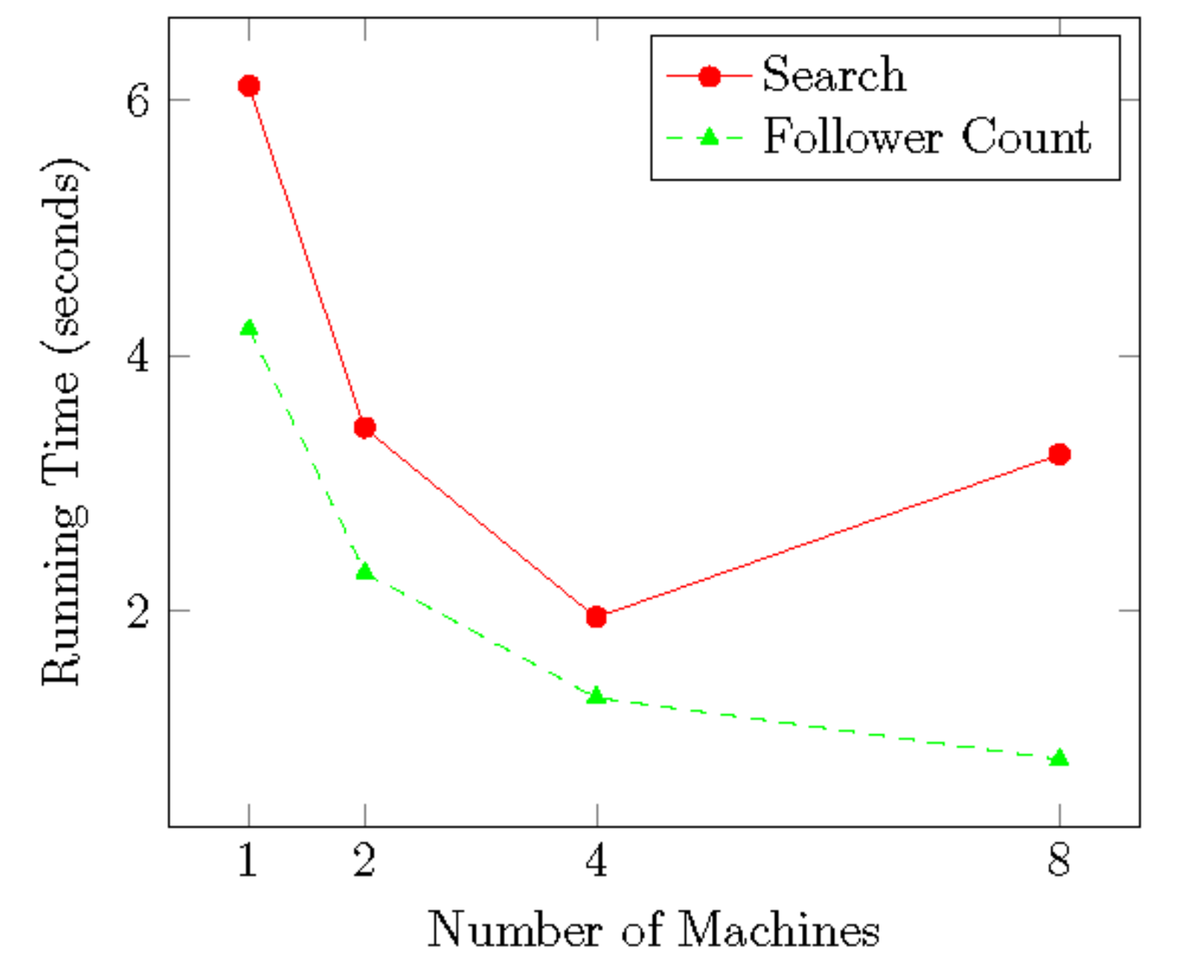
Motivation

- Real-time data analysis
  - Operates on recent and historical data
  - Collect and process data on the same node for low latency
  - Simplifies data collection
- Data processing
  - Faster to read and process data in parallel
  - Data processing can be done locally in most cases
- Scalability
  - Simple expansion of capacity and throughput by adding more nodes

Performance

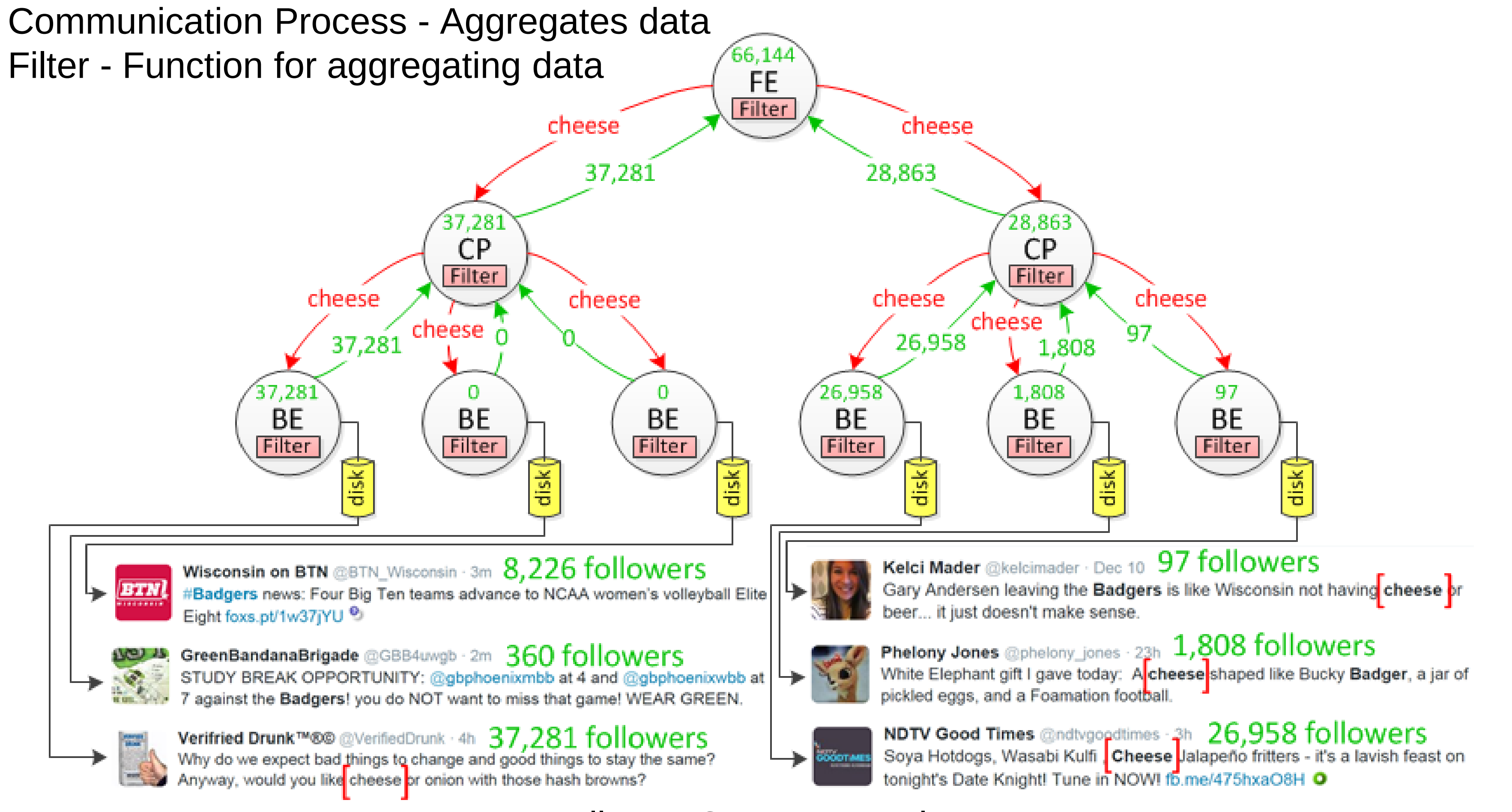
- Token passing
  - Measured data collection from a predictable data source
  - The cost of token passing is amortized as each node collects more data
- Data processing
  - Measured running time to process 10 million Tweets partitioned equally among various number of nodes
  - Follower count scales proportionately with the number of nodes
  - Search is optimal at 4 nodes due to bottleneck from limited network bandwidth





Data Processing Example

- MRNet
  - Frontend - Constructs tree and send keyword to backend
  - Backend - Reads disk, processes data, and sends results to frontend
  - Communication Process - Aggregates data
  - Filter - Function for aggregating data



Wisconsin on BTN @BTN_Wisconsin · 3m
#Badgers news: Four Big Ten teams advance to NCAA women's volleyball Elite Eight foxs.pt/1w37jYU

GreenBandanaBrigade @GBB4uwgb · 2m
STUDY BREAK OPPORTUNITY: @gbphoenixmbb at 4 and @gbphoenixwbb at 7 against the Badgers! you do NOT want to miss that game! WEAR GREEN.

Verifried Drunk™®® @VerifiedDrunk · 4h
Why do we expect bad things to change and good things to stay the same? Anyway, would you like cheese or onion with those hash browns?

Kelci Mader @kelcimader · Dec 10
Gary Andersen leaving the Badgers is like Wisconsin not having cheese or beer... it just doesn't make sense.

Phelony Jones @phelony_jones · 23h
White Elephant gift I gave today: A cheese shaped like Bucky Badger, a jar of pickled eggs, and a Foamation football.

NDTV Good Times @ndtvgoodtimes · 3h
Soya Hotdogs, Wasabi Kulfi , Cheese Jalapeño fritters - it's a lavish feast on tonight's Date Knight! Tune in NOW! fb.me/475hxaO8H

Follower Count Example

Bonsai: A Distributed Data Collection and Storage System for Data Stream Processing

Kritphong Mongkhonvanit and Kai Zhao
{kritphon, kzhao32}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin - Madison
1210 W. Dayton St.
Madison, Wisconsin 53706-1685 USA

Abstract

Bonsai is a distributed data collection and storage system for high volume data stream processing. Bonsai facilitates data analysis that requires large amounts of up-to-date data by automating the process of collecting and updating data. MRNet, a high-throughput communication system for parallel tools, is used to perform analysis on the collected data. Bonsai provides an additional abstraction for accessing incoming data stream as if it were static. We provide several examples involving analysis of a large set of concurrently arriving Tweets. Our result shows significant gain from distributing the analysis across multiple nodes while making no special considerations in the analysis program for the fact that data is arriving constantly.

1 Introduction

Many forms of data analysis rely on having access to a large amount of up-to-date data. However, spinning disks, which are the main storage medium for many systems, tend to be relatively slow in comparison to other parts of the system. Ousterhout [6] suggests that disk performance improvements has not kept up with its increase in capacity. This slowness can potentially become the bottleneck of the system, limiting throughput even when processing power is available.

Bonsai provides a solution to this problem by distributing data collection and storage to multiple machines, allowing high volumes of data to be collected, stored, and accessed efficiently. Distributing and parallelizing disk accesses across multiple machines allows the total bandwidth to greatly exceed that of a single machine. This design also offers simple capacity expansion by adding more machines to the *data collection ring*.

This paper focuses mainly on the design and architecture of Bonsai. Section 2 presents the Bonsai architecture, which is composed of the ring and the tree. Section 3 presents the design considerations taken into account when building the system. Section 4 presents some sample data analysis that can be done with

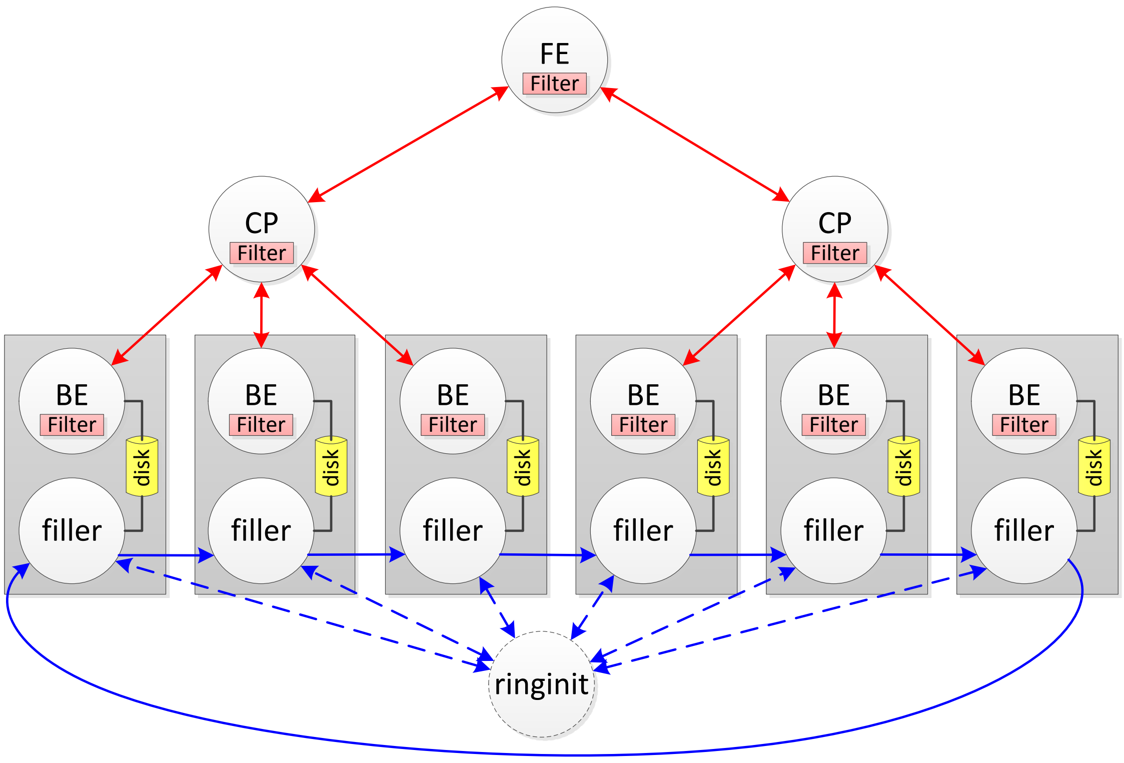


Figure 1: This diagram illustrates the architecture Bonsai, which consists of two main subsystems: the data collection ring and the data processing tree. The data collection ring, shown on the bottom in blue, consists of multiple data collection processes called *fillers* connected in a ring structure. The tree structure shown on top in red represents the data processing subsystem implemented using MRNet. It consists of 4 main components: frontend (FE), communication processes (CP), backends (BE), and filters. The direction of the arrows denotes the direction of communication. The data collection ring initialization server, *ringinit*, is drawn using dashed lines to show that it is not used after ring initialization.

Bonsai. Section 5 presents the performance of data collection and data analysis. Section 6 describes related works and section 7 concludes the paper.

2 Architecture

Bonsai consists of two subsystems: data collection and data processing. The data collection subsystem is responsible for gathering data from a source, which, in our case, is a down-sampled live stream of Twitter data stream. The collected data will then be processed by the data processing subsystem on-demand. These two subsystems operate independently, with no communication with each other except through the collected data.

2.1 Data Collection

Each node runs an instance of the data collection process referred to as *filler*. A single instance of *ringinit*, the address distribution server, is used to assist *fillers* in connecting to each other. The connected *filler* will form a ring structure.

The ring initialization process starts with the *fillers* informing *ringinit* of their presence and their port numbers. The *fillers* will then proceed to send a request to *ringinit* for the list of all *filler* nodes. *ringinit* will respond with a wait command until all fillers have contacted *ringinit*. Upon receiving this wait command, the fillers will wait for an amount of time – one second in our implementation – before contacting *ringinit* again.

When all *fillers* have contacted *ringinit*, the server will respond to the *filler*'s request with the list of addresses and ports of all fillers. Upon receiving this information, the *filler* will independently determine the next *filler* instance to pass the token to. This is done simply by choosing the node immediately after *filler*'s own entry in the list, wrapping around if necessary. The effect of this algorithm is that every *filler* will have exactly one *filler* to pass the token to and one *filler* to receive the token from, which results in a ring structure among the nodes. *ringinit* plays no part in the actual operation of the *filler* ring and can be shut down after initialization phase with no negative effects.

After ring initialization, *ringinit* will assign the initial token to the first *filler* to contact *ringinit* during ring initialization. This token is used to decide which *filler* will be collecting data. The *filler* that has the token will start collecting data from the stream until its buffer file becomes full. Afterwards, the token will be passed to the next *filler* instance. This process repeats until all nodes within the ring has filled its buffer file.

After the last *filler* has filled its buffer file, it will pass the token to the node that has been assigned the token initially. Upon receiving the token, that node will start replacing its buffer file on an entry-by-entry basis until the file is completely overwritten. Afterwards, the token will once again be passed to the next node. This process repeats until the Bonsai ring is shutdown.

Each *filler* stores its data in a buffer file, which is just a file with a fixed data record size and a fixed number of records. These parameters can be adjusted to accommodate different use cases. This design allows the data to be replaced in an efficient manner – the fixed size records allows fast lookup, and the fixed file size reduces file fragmentation.

Simple recovery mechanism can be implemented by having *filler* create an actual file to represent the token received. Through this, *filler* can inform *ringinit* whether it was the token holder before failure. *ringinit* can then assign the initial token to that node so that the filling process will proceed from the point of failure. In addition, filler reads the time stamp of each data record upon starting to find the index of the oldest data to start replacing. However, this feature is not fully implemented due to time constraints.

2.2 Data Processing

Data processing is done using MRNet [1], a high-throughput communication system that highly simplifies the process of building distributed programs. The basic architecture of the system comprises comprises of a frontend, backends, communication processes, and filters. The frontend constructs the MRNet tree and starts the backends. The backends performs computation on collected data and send results to the frontend. Communication processes may be used to filter the results from a large number of backends more efficiently. The filter is a user-defined reduction function used to aggregate results from the backends. In our system, the backends mainly access data from its local disk to minimizes network communication and increase performance.

This part of the system can be customized to perform various computations. MRNet programs can be written to run on Bonsai without making any special accommodations for it; collected data can be accessed just like normal files. Because of this, it is possible to run data processing programs written for Bonsai even in absence of the data collection subsystem.

3 Design Considerations

We considered many platforms on top of which we can implement the data processing subsystem. An option we considered is Google MapReduce [2]. MapReduce requires data to be copied over to GFS [4], which increases the latency to process incoming Tweets. Because of this latency, it is unsuitable for real-time data analysis. Furthermore, copying data to GFS will require twice the amount of storage.

Spark [5] is another option we considered. It attempts to speed up computations by pre-loading data into main memory. However, this mechanism is unlikely to be effective in our case because we are constantly overwriting old data, causing data in main memory to be stale and inconsistent with data on disk. Moreover, loading data to main memory would incur large amount of delay, which would be undesirable for interactive applications such as real-time search.

We ended up choosing MRNet [1] because of its flexibility. MRNet provides a simple communication framework for sending data to and retrieving data from worker nodes. MRNet does not impose any restrictions on the kind of data store on which it operates, making it suitable for use with our custom data store.

We store data in a file that has fixed number of fixed-size records, ensuring that the file size remain constant. This design offers fast access to arbitary records at the cost of larger file sizes. An alternative implementation is to use variable-sized records. While this approach would be more efficient in terms of storage, accessing specific records can be slow since that would require reading the contents of the file from the beginning. The file may also need to be resized as new data is collected, which can result in a high amount of file fragmentation. Since Tweet records are limited to a few hundred bytes, we determined that the first approach fits our use case better. However, if Bonsai is used with data set that has a larger data record size, then variable-sized records might be a better design decision.

Updating filled buffer file is done by overwriting data records in-place rather than filling a new temporary buffer file that will replace the original file when it is fully filled. Using a temporary buffer would require twice as much storage space than overwriting data records in-place because two buffer files have to exist when *filler* is active. This can be problematic with large amounts of data per node.

4 Examples

We implemented two example data processing applications to demonstrate some use cases that would be suitable for Bonsai. To measure the performance of the system, these examples are also used for benchmarking.

Example 1. Follower count indirectly approximates the popularity of given keywords. The frontend takes one or more keywords as input and sends the them to the backends. The backends then scan their disk for all Tweets containing the keywords and extract the follower count of the users who created those Tweets. This number is summed up and sent to the frontend, where the results from all backends are summed up and printed out. The result is the approximate number of Twitter accounts that are exposed to the keyword.

Example 2. Search works similarly to how search works on Twitter website. The frontend takes one or more keywords as input and sends them to the backend. The backends then scan the collected Tweets data for all Tweets containing at least one of the keywords and send all matching Tweets to the frontend, where the results are aggregated and printed out.

5 Performance

5.1 Data Collection

We first measure data fill rate with respect to the number of nodes in the data collection ring. The buffer is filled by reading blocks of size 4096 bytes from `/dev/urandom`. Tests were then performed with varying numbers of blocks read per filler, specifically 128, 1024, and 8192. We chose to read random numbers rather

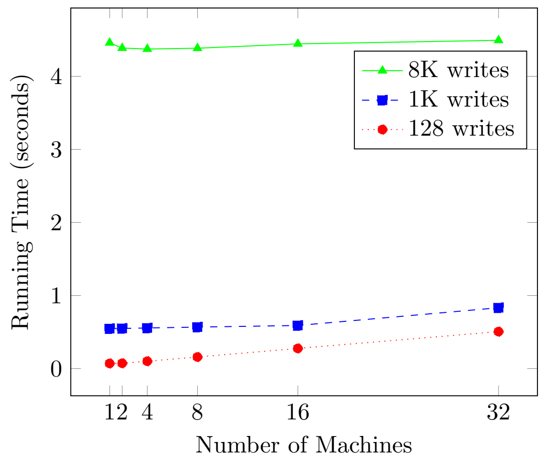


Figure 2: Running time to collect data versus number of machines in the data collection ring.

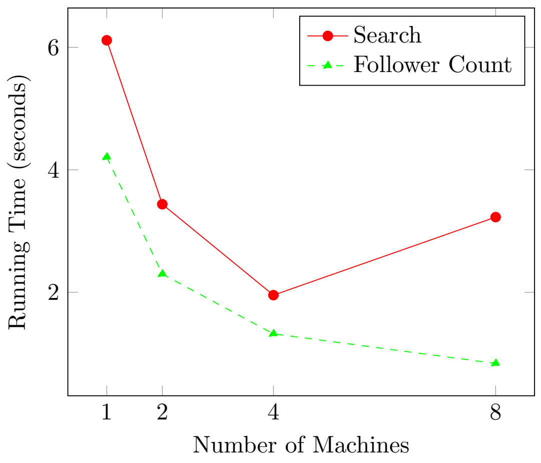


Figure 3: Running time to process 10 million Tweets versus number of backends nodes in the tree.

than actual Twitter stream because the rate at which the Tweets arrive is not guaranteed to be constant. There are many factors that may affect the stream rate, from the number of people posting Tweets to the level of congestion in our network. Reading from `/dev/urandom` eliminates most of these external uncertainties which makes it a much more predictable data source.

The rate at which the time taken to fill the buffer increases as the number of nodes in the ring increases is noticeably higher with lower number of reads compared to that of higher number of reads. This suggests that the cost of passing token around the ring is amortized when the buffer size on each node is sufficiently large.

The results shown in figure 2 shows that the filler is able to do 8K reads of 4 KB blocks in 4.5 seconds. Extrapolating from this information, the filler is able to collect 614 GB of data per a day. Based on collected Tweets, each Tweet takes about 500 bytes of data to store. Since there are around 500 millions Tweets – or about 250 GB of data – posted per day [9], our system can support the full twitter stream rate. If we need even greater collection rates, then we can use multiple rings or multiple tokens with some mechanism to avoid data duplication. However, this mechanism is not implemented yet due to time constraints.

5.2 Data Processing

We ran the follower count example and the search example on 10 million Tweets. No keywords were passed to follower count, which aggregates the number of followers of posters of all Tweets in our data set. Search was tested with the string ":)" used as the keyword. The benchmarking results show that the running time for follower count scales proportionately with the number of nodes whereas the running time for search is at its minimum at 4 processing nodes. This is because our searching algorithm is computationally light while generating a relatively large amount of output. This amplifies the cost of returning the result back to the frontend with larger number of nodes, causing the network bandwidth to become the bottleneck of the system.

6 Related Works

The paper by Olston presents Nova [3], a workflow manager for a continuous stream of data. Nova combines the results from previous requests with the results from the continuous data stream. Therefore, Nova is able to handle an theoretically infinitely large stream of data by storing just the results instead of the the raw data. For example, Nova can store the results of word count on a large dataset by storing one copy of each word and its respective count.

The paper by Lam presents Muppet [7], a MapReduce-style of processing fast data. Like Nova, Muppet also combines the results from previous requests with the results from the continuous data stream. However, unlike Nova, Muppet stores the results in a key-value store for higher efficiency, scalability, and availability. Both Nova and Muppet differs from our proposed work because our proposed work will keep the raw data to support non-iterative MapReduce tasks.

The paper by Keleher presents Treadmarks [8], a distributed shared memory for storing large data structures in cache. Treadmarks support parallel computing by distributing storage, allowing each machine to map local data. The results can be aggregated using the shared address space. Treadmarks differs from our proposed work because Treadmarks has no mechanisms for replacing old data.

7 Conclusion

Bonsai simplifies the process of collecting, storing, and updating continuous data stream for data analysis with minimal impact to the analysis program. The data collection benchmark shows that performance penalty of the data collection ring is minimal and will be amortized as the amount of data collected per node increases. The data processing benchmark likewise shows visible gains on the performance as the number of processing nodes increases.

Though Bonsai have achieved our preliminary goals, there are still many areas that can be improved. One such area is fault tolerance. Currently, Bonsai is not able to handle *filler* failure at all; failure of a single *filler* will cause the whole data collection ring to stop. We plan to implement a mechanism similar to *hinted hand off* to allow *fillers* to automatically detect and skip failing nodes. Another aspect that can be improved is dynamic data collection ring reconfiguration. Reconfiguring the ring currently requires a full restart of the ring. A mechanism to allow adding and removing *fillers* after the collection process has started would significantly ease the administration of the system. Our future research will focus largely on addressing these issues, which will improve the overall usability and reliability of the system.

References

[1] P. Roth, D. Arnold, and B. Miller. "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools". Supercomputing, November 15-21, 2003.

[2] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". 6th Symposium on Operating Systems Design and Implementation, pages 137-149, 2004.

[3] C. Olston , G. Chiou , L. Chitnis , F. Liu , Y. Han , M. Larsson , A. Neumann , V. B.N. Rao , V. Sankarasubramanian , S. Seth , C. Tian , T. ZiCornell , and X. Wang. "Nova: continuous Pig/Hadoop Workflows". Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, June 12-16, 2011, Athens, Greece.

[4] S. Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System".Symposium of Operating Systems Principles, October 19-22, 2003, Bolton Landing, New York, USA.

[5] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster Computing with Working Sets". Hotcloud, 2010.

[6] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazieres, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. "The Case for RAMCloud". Communications of the ACM,2011.

[7] W. Lam, Lu Liu, STS Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. "Muppet: MapReduce-Style Processing of Fast Data". The 38th International Conference on Very Large Data Bases, August 27-31, 2012.

[8] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems". Proceedings of the USENIX Winter 1994 Technical Conference, 1994.

[9] S. Kim. "Twitter's IPO Filing Shows 215 Million Monthly Active Users". ABC News. `http://abcnews.go.com/Business/twitter-ipo-filing-reveals-500-million-tweets-day/story?id=20460493`