Syndesis: Mapping Objects to Files for a Unified Data Access System

Anthony Kougkas - akougkas@hawk.iit.edu, Hariharan Devarajan, Xian-He Sun

What is this talk about?

- Highlights of this work:
  - Key characteristics of object-based and file-based storage systems.
  - Design and implementation of a unified data access system that bridges the semantic gap between object-based and file-based storage systems.
  - Evaluation results show that, in addition to providing **programming convenience and efficiency**, our library, **Syndesis**, can grant **higher performance** avoiding costly data movements between object-based and file-based storage systems.

Different communities - different systems

The tools and cultures of HPC and BigData analytics have diverged, to the detriment of both; **unification** is essential to address a spectrum of major research domains.

- D. Reed & J. Dongarra

<u>Syndesis: Unified Data Access System</u>

Challenges of storage unification

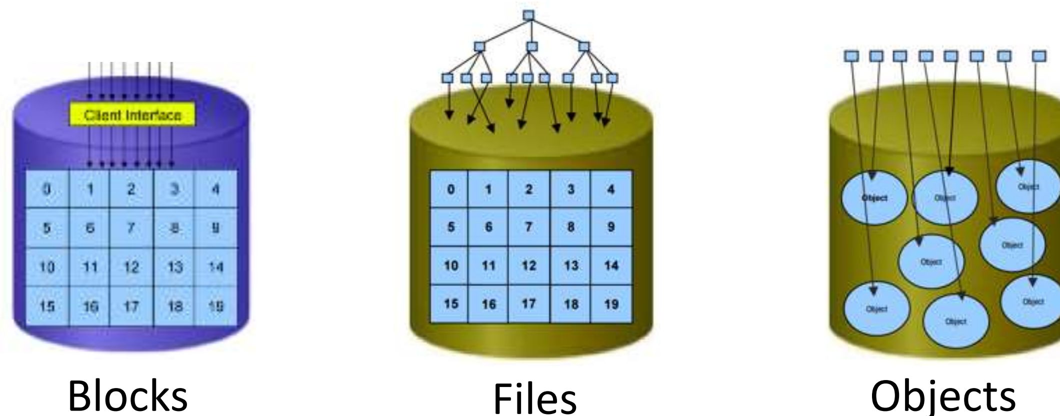
- Wide range of issues:
  1. There is a gap between
     a) **traditional storage solutions** with semantics-rich data formats and high-level specifications, and
     b) **modern scalable data frameworks** with simple abstractions such as key-value stores and MapReduce.
  2. There is a big difference in architecture of programming models and tools.
  3. Lack of management of
     1. heterogeneous data resources
     2. diverse global namespaces stemming from different data pools.

Our thesis

- A radical departure from the existing software stack for both communities is not realistic.
- Future software design and architectures will have to raise the abstraction level and therefore,
  - **bridge the semantic and architectural gaps.**
- We envision
  - a data path agnostic to the underlying data model
  - leverage each storage system's strengths while complementing each other for known limitations.

Data formats and storage systems

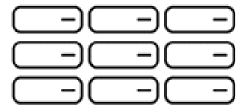
- Data are typically represented as files, blocks, or objects.
- Two major camps of storage solutions:
  - File-based storage systems
    - POSIX-I/O with fwrite(), fread(), MPI-I/O with MPI_File_read(), MPI_File_Write()
    - High-level I/O libraries e.g., HDF5, pNetCDF, MOAB etc
  - Object-based storage systems
    - REST APIs, Amazon S3, OpenStack Swift with get(), put(), delete()



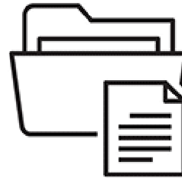
Interface and API

- Storage expectations:
  - MPI and scientific computing
  - Hadoop ecosystem and BigData computing
  - POSIX compliant or not?
  - Structured, semi-structured, and unstructured data
  - Consistency models: strong vs eventual?

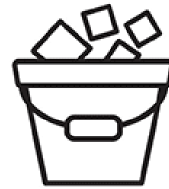
Data models differences



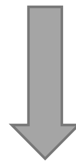
Block storage
Data stored in fixed-size 'blocks' in a rigid arrangement—ideal for enterprise databases

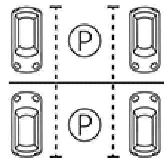


File storage
Data stored as 'files' in hierarchically nested 'folders'—ideal for active documents

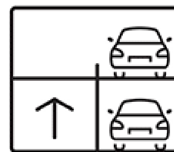


Object storage
Data stored as 'objects' in scalable 'buckets'—ideal for unstructured big data, analytics and archiving

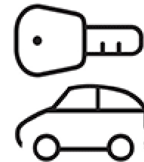




Block storage
'Parking lot' metaphor—data stored in rigidly defined blocks—access by specific 'space' location



File storage
'Parking garage' metaphor—data arranged in hierarchical levels—retrace path to access



Object storage
'Valet parking' metaphor—no need to worry about storage details—easy to store and access data

Source: Dell EMC

Category	Object Storage	File Storage
Data unit	Objects	Files
Update	Create new object	In-place updates
Protocols	REST and SOAP	NSF with POSIX
Metadata	Custom	Fixed attributes
Strengths	Scalability	Simplified access
Limitations	Frequent updates	Heavy metadata
Performance	High throughput	Streaming of data

- There is no "one-storage-for-all" solution.
- Each system is great for certain workloads
- Unification is essential

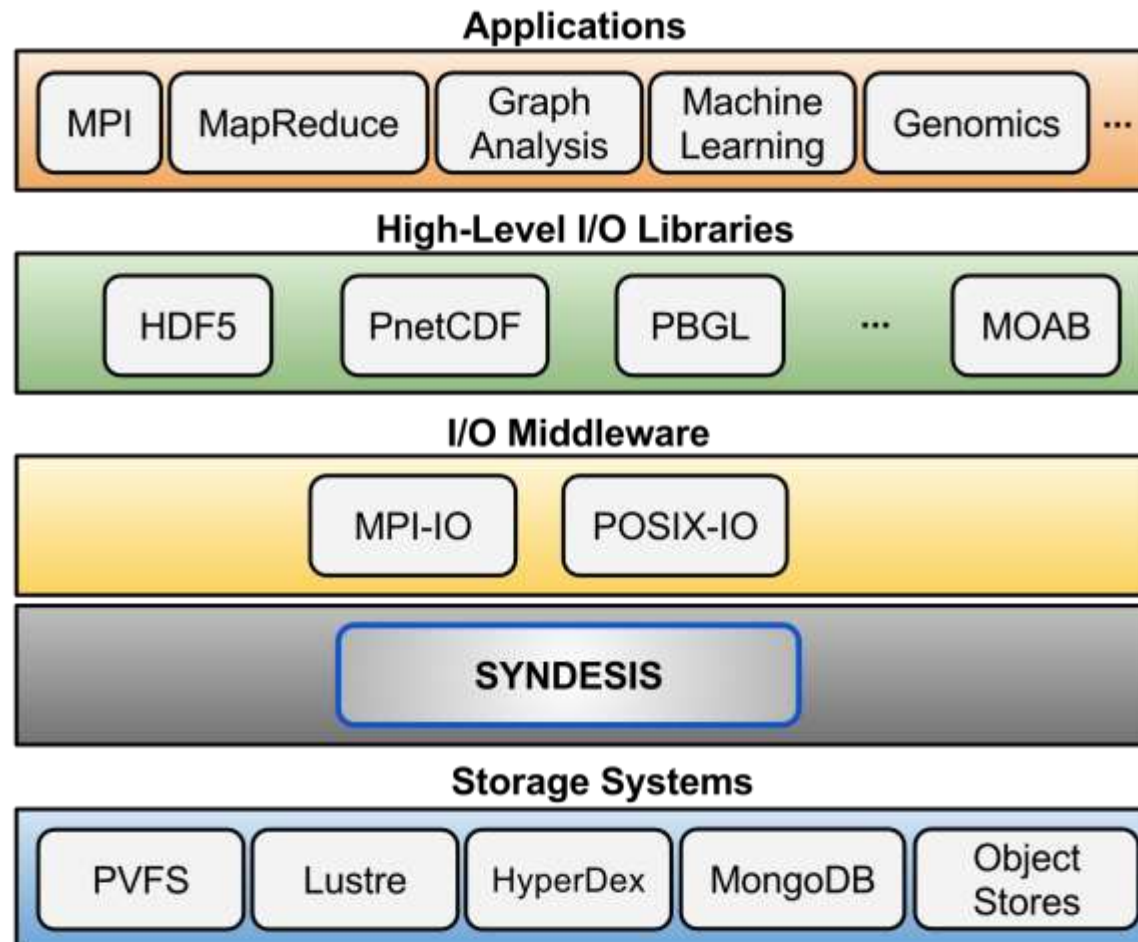
Related work

- From the File system side:
  - CephFS
  - PanasasFS
  - OBFS: A File System for Object-based Storage Devices OSD

- From the Object store side:
  - AWS Storage Gateway
  - Azure Files and Azure Disks
  - Google Cloud Storage FUSE

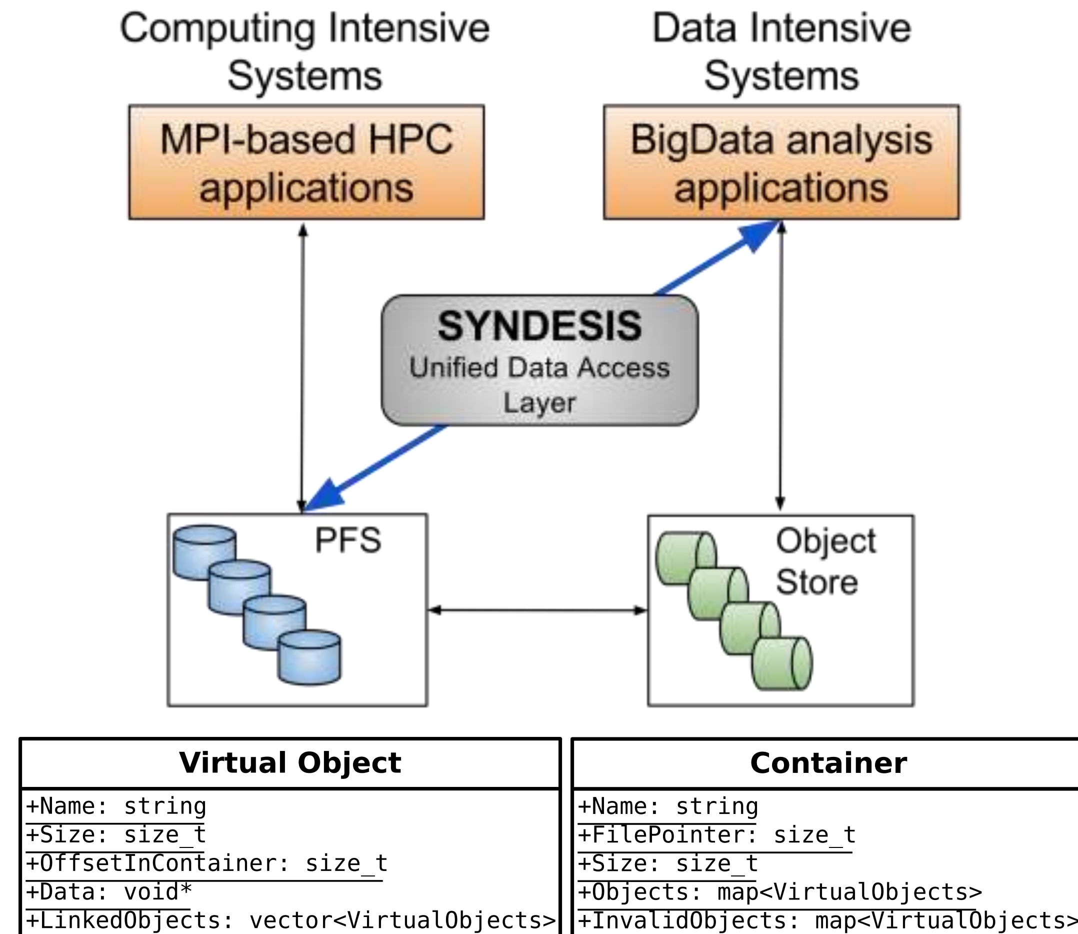
Syndesis is a general solution that can bridge any Object Store with any File System and does NOT require changes in user code or underlying system deployments.

Design



- Middle-ware library with modular design, written in C++
- Link with applications (i.e., re-compile or LD_PRELOAD)
- Existing files are loaded upon bootstrapping via crawlers
- Metadata in memory for faster lookups
- Deletions via invalidation
- KeySpace cleanup via garbage collection upon termination
- Syndesis does not require any changes to the application code but instead intercepts all storage calls (e.g. CRUD operations) and redirects them to a file system.

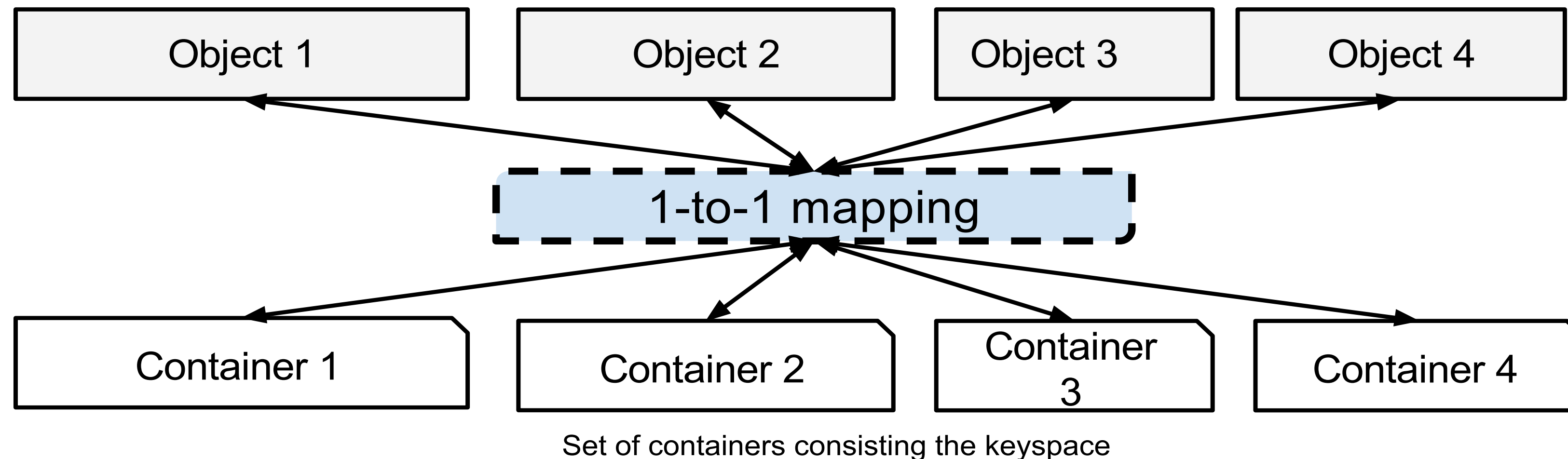
Design



- Four new mapping strategies:
  1. 1-to-1
  2. N-to-1
  3. N-to-M (simple)
  4. N-to-M (optimized)
- *Virtual Object*: encapsulates the application's object along with other metadata information necessary to our library. It is the unit of mapping to the underlying file system.
- *Container:* represents a file in the file system that holds virtual objects and other metadata information useful to our library such as indexing and update logs.

Design: 1-to-1 Mapping

- Each object is mapped to a unique container.
- Ideal for accessing existing datasets.
- Good performance for relatively small number of objects.



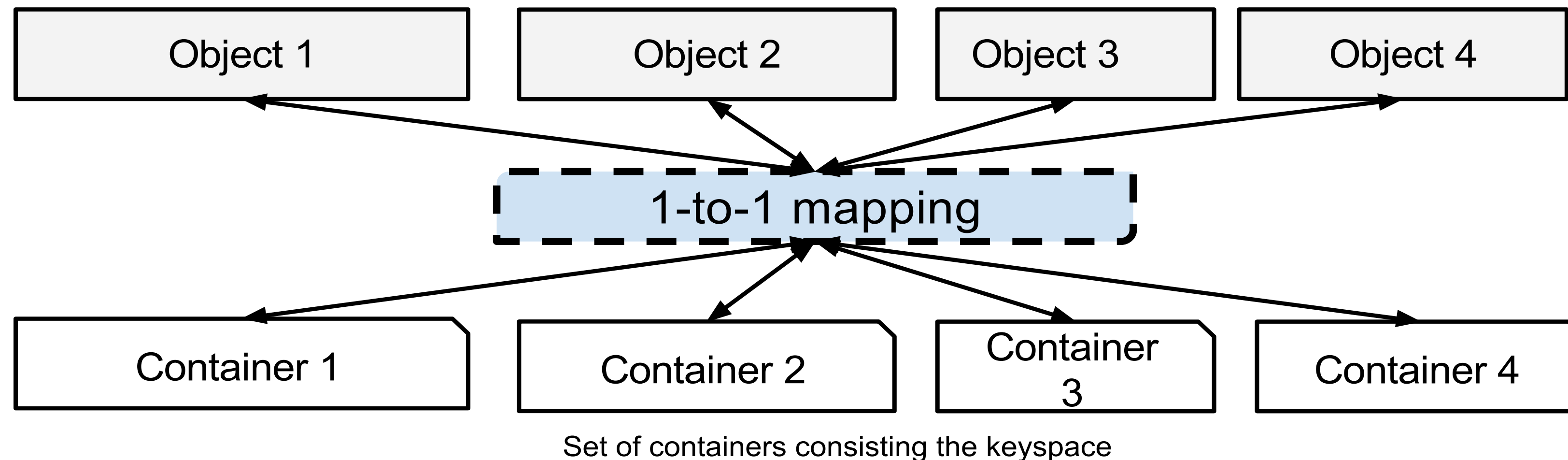
Design: 1-to-1 Mapping

Benefits:

- Simpler mapping semantics
- Less mapping and memory overhead
- Faster update operations

Limitations:

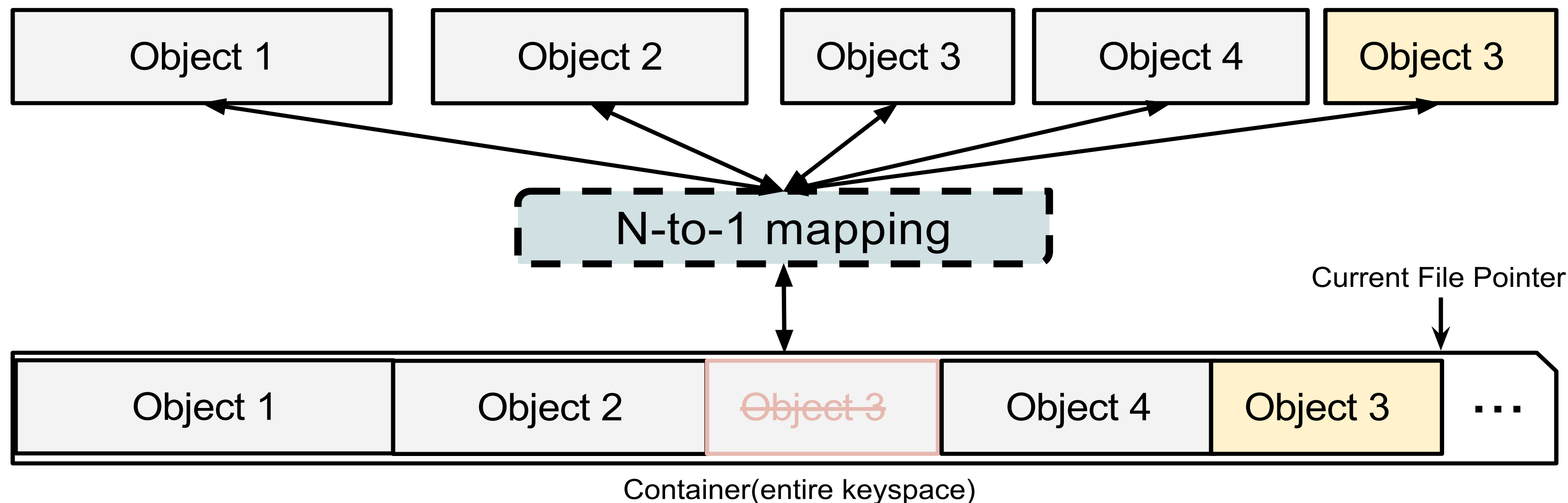
- Number of files and opened file handlers is relatively small.
- Low performance with large number of files
- Not scalable



Set of containers consisting the keyspace

Design: N-to-1 Mapping

- Entire keyspace is mapped to one container. Virtual objects are written sequentially.
- Updates are simply appended at the end of the file while invalidating the previous object.
- Indexing is important for faster get() operations.



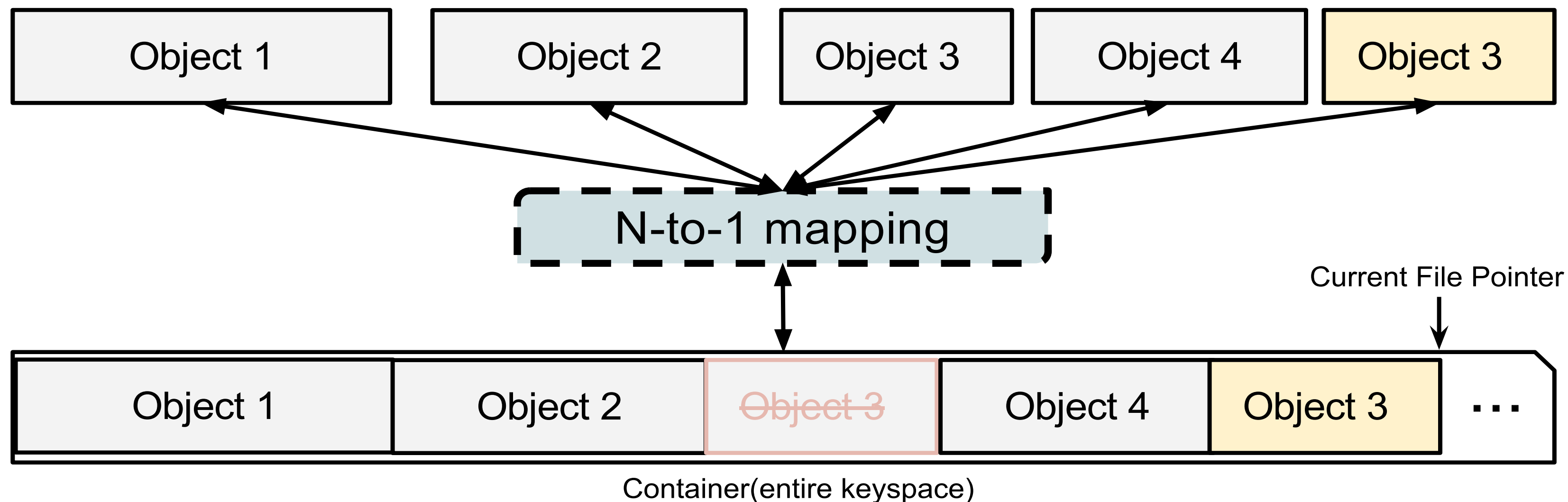
Design: N-to-1 Mapping

Benefits:

- Light metadata (i.e., only one container)
- Faster searching (i.e., querying)
- Mapping cost is relatively low

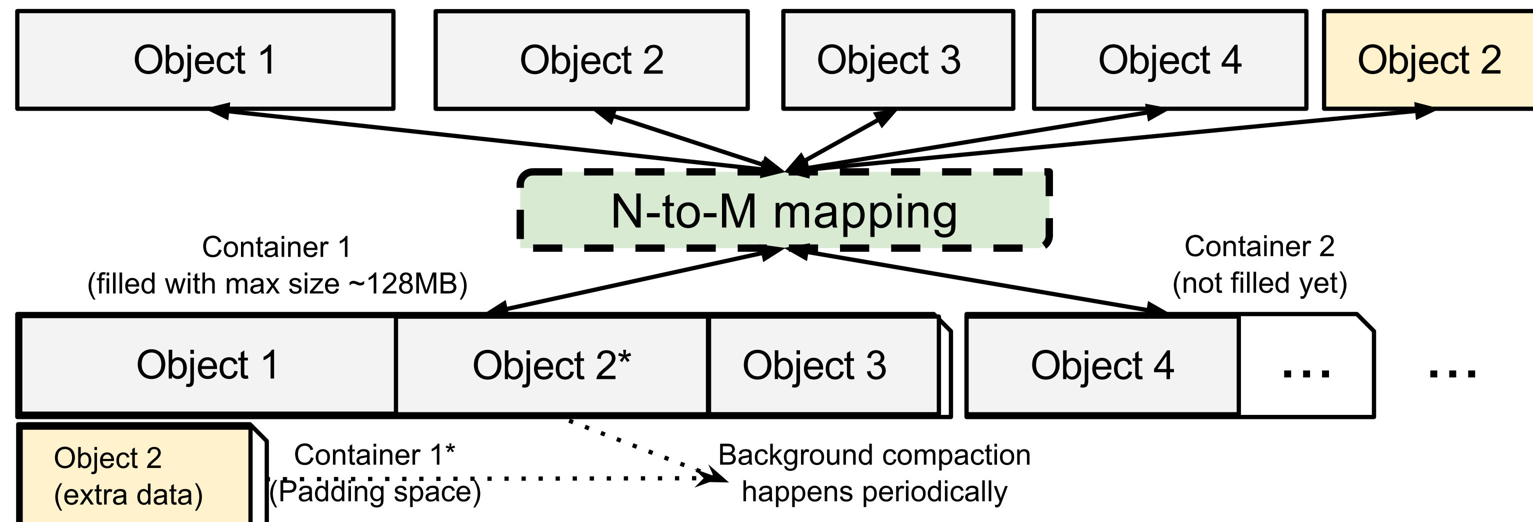
Limitations:

- Container size cannot grow infinitely
- Fragmentation of the container
- Limited scalability



Design: N-to-M (Simple) Mapping

- A collection of objects is mapped to a collection of containers.
- Threshold to create new containers (default every 128MB) bounding the total number of containers.
- Special container-> update container for padding



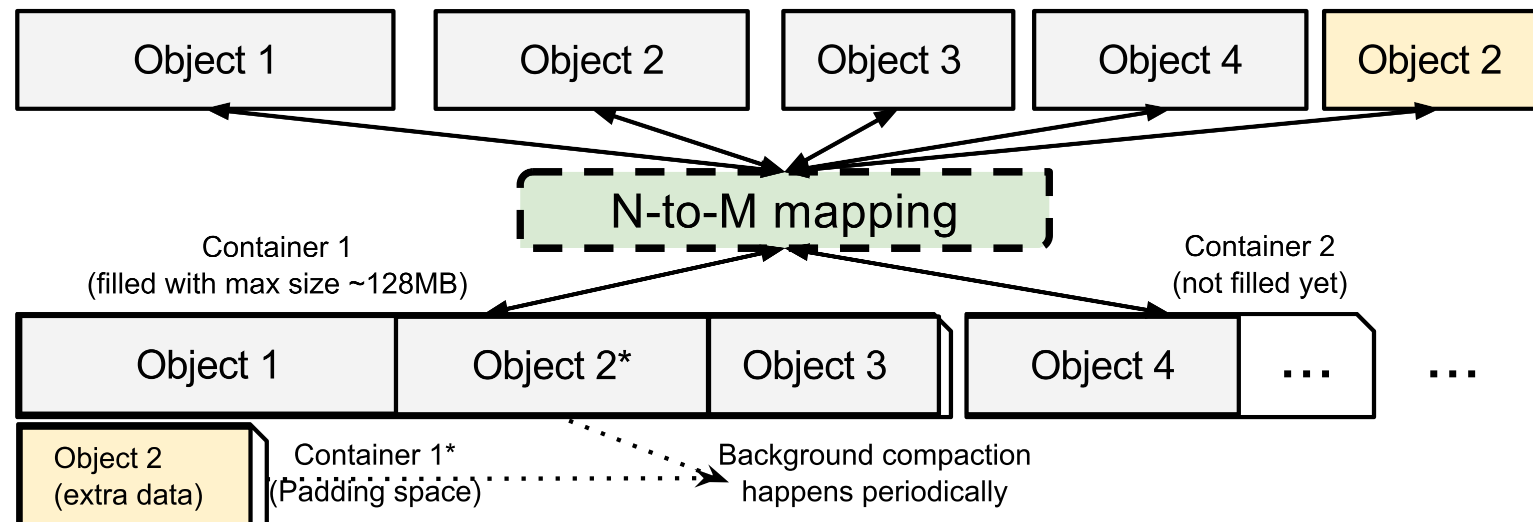
Design: N-to-M (Simple) Mapping

Benefits:

- Favorable access patterns to the underlying file system (~128MB)
- Great write performance
- Cached containers help read operations

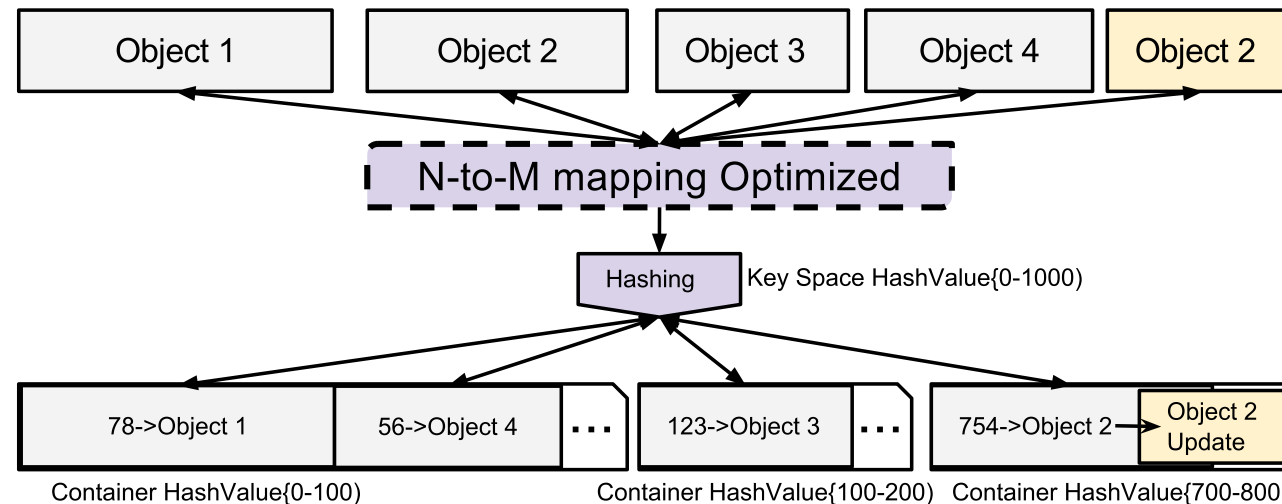
Limitations:

- Tricky updates and searching
- Mapping cost is higher
- Background compaction can be expensive



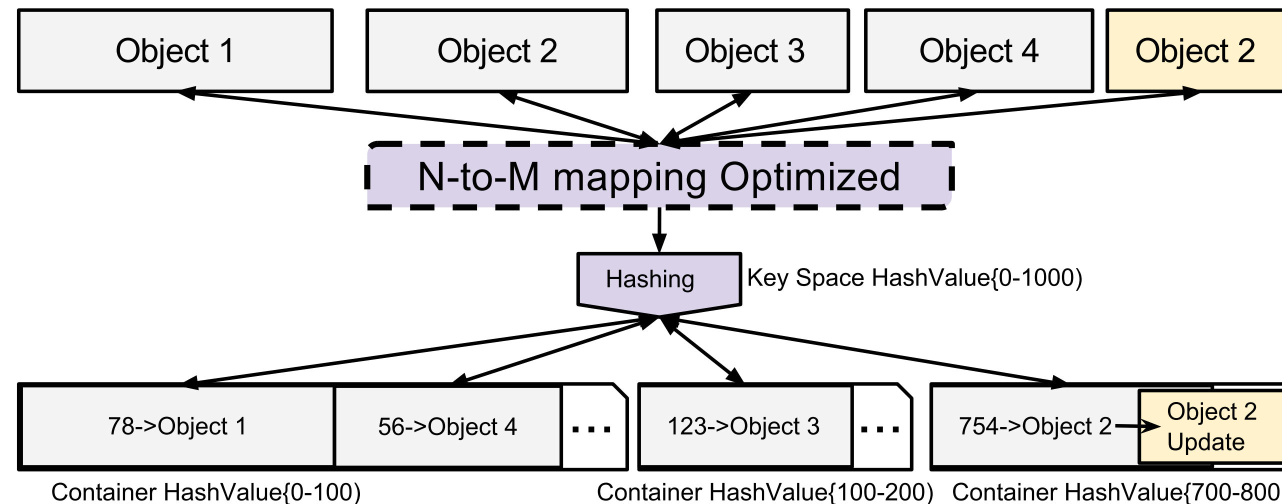
Design: N-to-M (Optimized) Mapping

- Objects are first hashed into a key space and then mapped to the container responsible for that range of hash values.
- Containers are created according to a range of hash values and their size is flexible (e.g., equal to the size of the keys it holds).
- Update operations simply write at the end of the container while invalidating the previous object.



Design: N-to-M (Optimized) Mapping

- Periodic container defragmentation to save storage space.
- Constant time searching via truth arrays.
- Faster non-blocking I/O using MemTables allows overlapping with computation



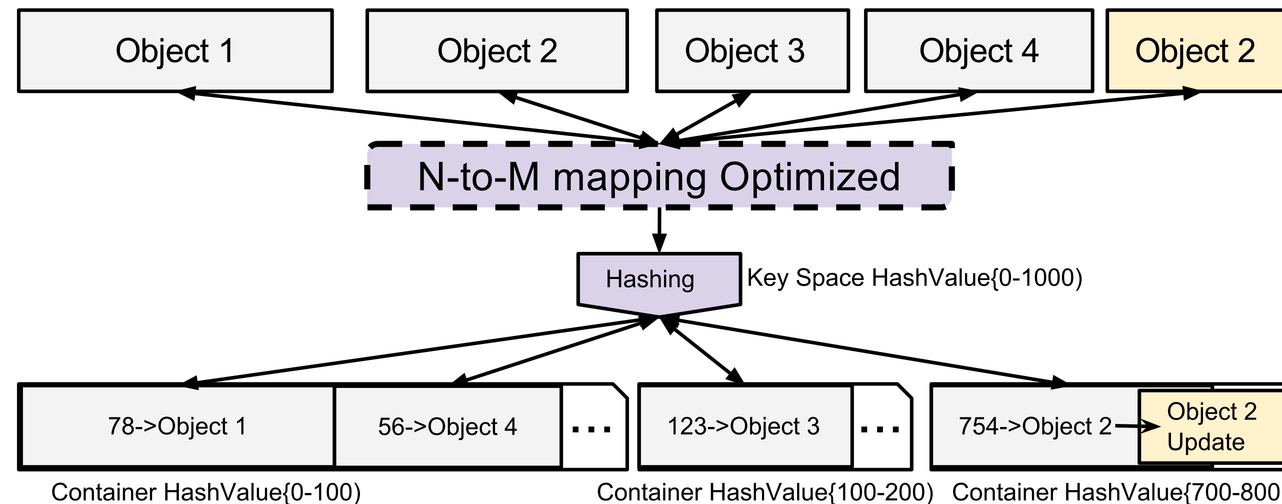
Design: N-to-M (Optimized) Mapping

Benefits:

- Extremely scalable
- Ideal for large number of objects
- Fast read and write operations

Limitations:

- Higher memory footprint
- Frequent background operations for defragmentation and memtable flushing
- Absolutely nothing else!!!!! ☺

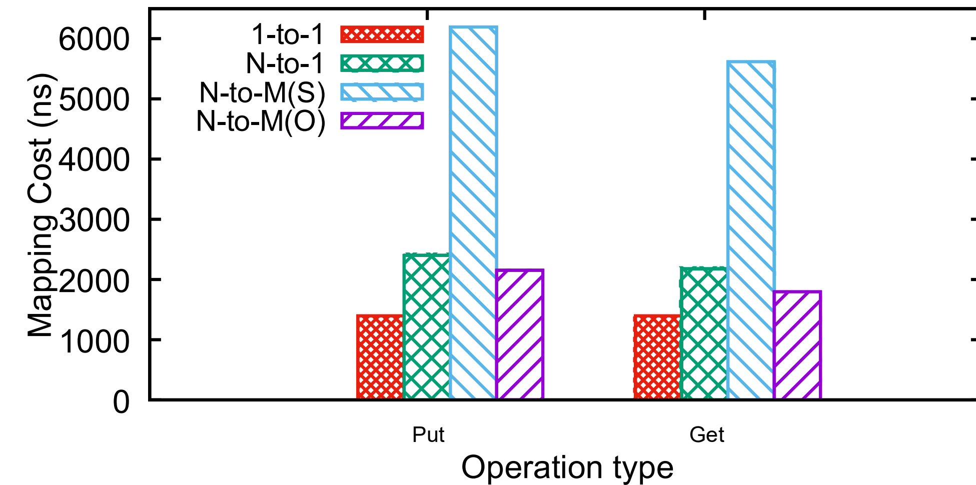


Evaluation Methodology

- Testbed: Chameleon System
- Appliance: Bare Metal
- OS: Centos 7.1
- Storage:
  - OrangeFS 2.9.6
  - Redis 4.0.1
- MPI: Mpich 3.2
- Programs:
  - Synthetic benchmark
  - YCSB
  - K-means clustering

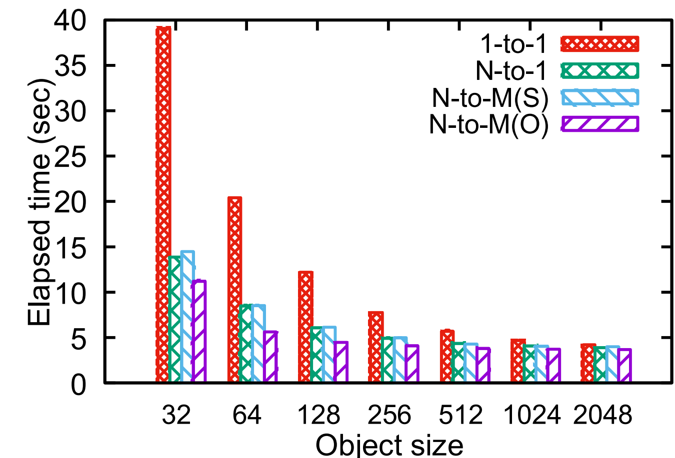
Evaluation Results – Library Overhead

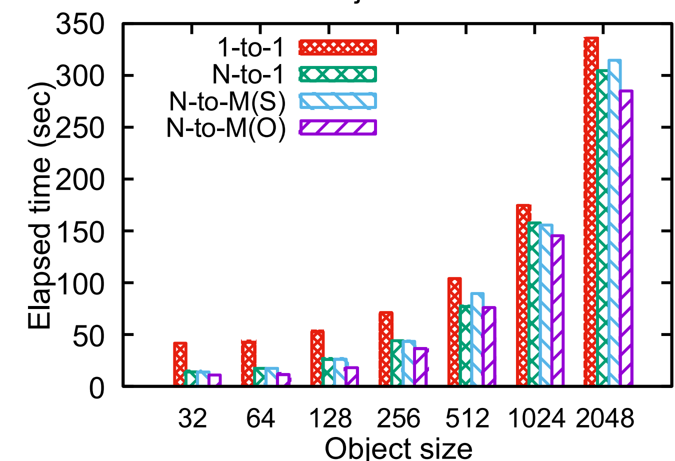
- Input: 128K objects of 64KB
- Output: Average time spend in mapping in ns (per operation)
- 1-to-1 simplest with lower cost
- N-to-M-Optimized only 2000ns
- Overheads kept minimum
  - 0.0050 - 0.0080% of the overall execution time (Mapping time over I/O time)



Evaluation Results – Virtual Object Size

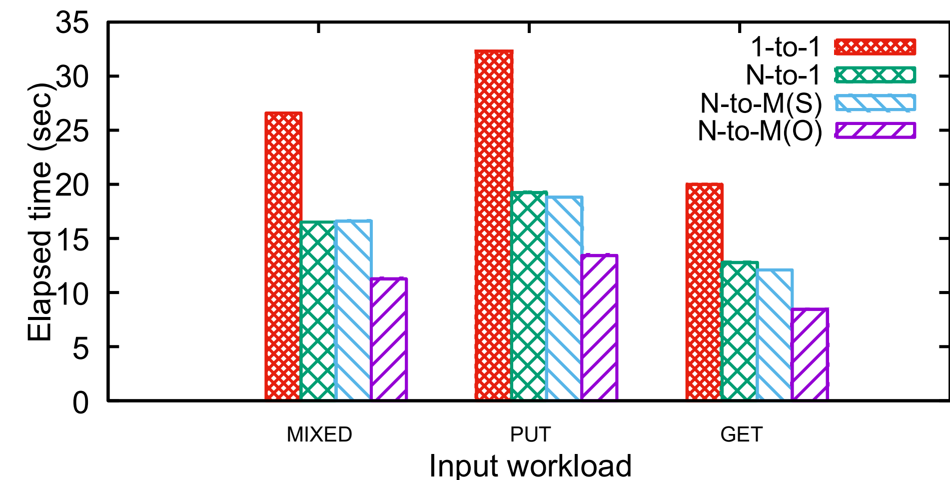
- Virtual Object is tunable
  - 32KB – 2MB
- Weak scaling (top figure) total I/O 1.2GB:
  - 40000 virtual objects of 32KB each
  - 600 virtual objects of 2MB each
- Strong scaling (bottom figure) total 40000 objects:
  - 1.2GB of 32KB virtual objects
  - 80GB of 2MB virtual objects
- Output: Execution time in seconds
- Best performance:
  - Bigger virtual objects (i.e., 2MB)
  - N-to-M Optimized strategy





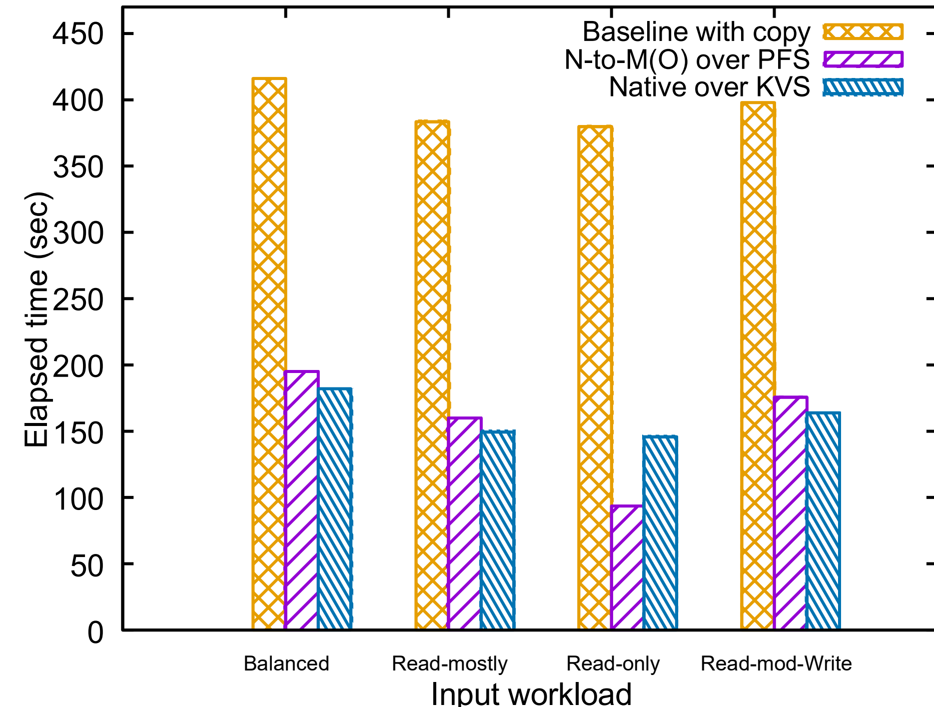
Evaluation Results – Benchmark

- Workload: 4GB total I/O
- Object size: 64KB
- Flow for mixed:
  - 1GB write followed by 1GB read
  - 1GB update followed by 1GB read
- Output: Overall execution time in seconds
- 1-to-1 suffers from large number of files
- N-to-M-Optimized performed more than **2x faster**



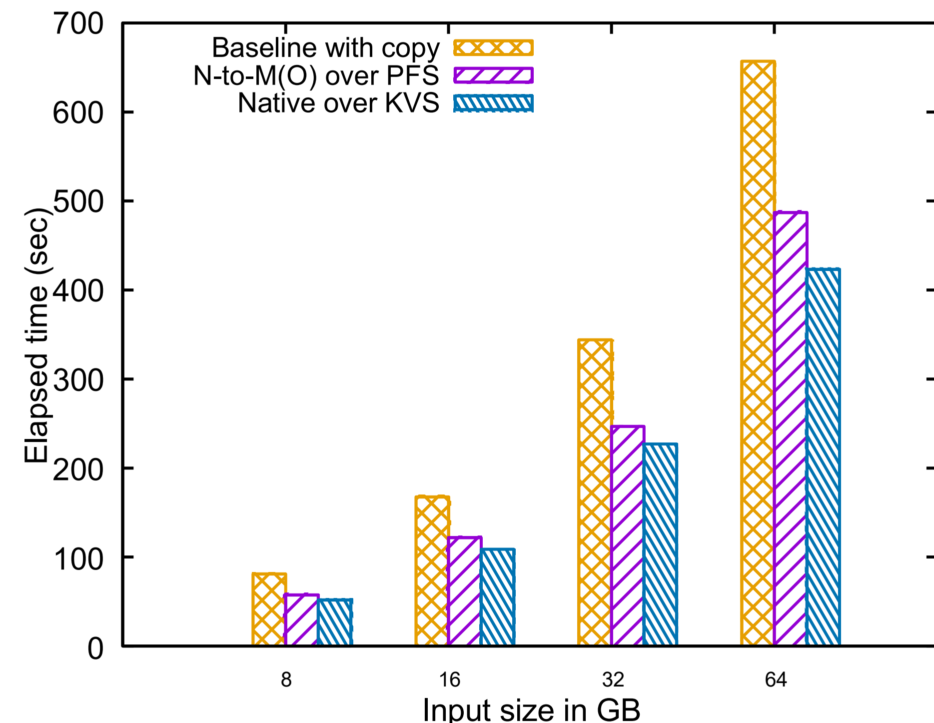
Evaluation Results – YCSB

- Offline data preloading
- Workloads:
  - Balanced: 50% reads and 50% writes
  - Read-mostly: 90% read and 10% writes
  - Read-only: 100% read
  - Read-modify-write
- Total I/O 64GB in 64KB objects
- Baseline flow:
  - Data are copied into the Redis and then run the test natively
- More than **2x speedup** with our solution, N-to-M-Optimized



Evaluation Results – YCSB

- Offline data preloading
- Input sizes:
  - 8, 16, 32, 64GB
- Baseline flow:
  - Data are copied into the Redis and then run Kmeans natively
- Minimal overhead
  - Less than 9%
  - 57 sec over 52 sec natively for 8GB
- More than **40% speedup** with our solution, N-to-M-Optimized



Conclusions & Future Steps

- File-based vs Object-based Storage solutions
- Four new algorithms to map objects to files
- Evaluation shows:
  - Minimal mapping overheads
  - More than 2x in performance speedup over naïve mappings
- Future work:
  - A new I/O management framework that integrates different storage subsystems and thus, get us closer to the convergence of HPC and BigData.
  - A system that offers universal data access regardless of the storage interface. Our mappings are a great first step ☺

Questions

Anthony Kougkas

akougkas@hawk.iit.edu

рахмат danke 謝謝 ngiyabonga
Баярлалаа spas teşekkür ederim
mersi kia ora barka welalin tack mahalo tapadh leat
спасибо faafetai lava vinaka спасибі blagodaram dank je misaotra matondo paldies grazzi
kiitos dankie nanni nandri dhanyavad maururu köszönöm thank you gracias хвала asante manana obrigada tenki
enkosi bedankt bayarlalaa gràcie hvala aciü akun dankon djiere dieuf tau mochchakkeram murakoze chokrane
дякую mamnun
dziękuje chnorakaloutioun gratias ago gràcies sulpáy taiku go raibh maith agat
sobodi děkuji sağolun sukriya kop khun krap
obrigado mèsi didi madloba najis tuke ありがとう tanemirt grazie arigatô takk dakujem trugarez
kam sah hamnida terima kasih rahmet rahmat ευχαριστώ diolch dhanyavadagalu shukriya merce мерси
তোমাকে ধন্যবাদ 감사합니다 xièxie merci