SDC
STORAGE DEVELOPER CONFERENCE
SNIA ■ SANTA CLARA, 2014

Data De-duplication for Distributed Segmented Parallel FS

Boris Zuckerman & Oskar Batuner

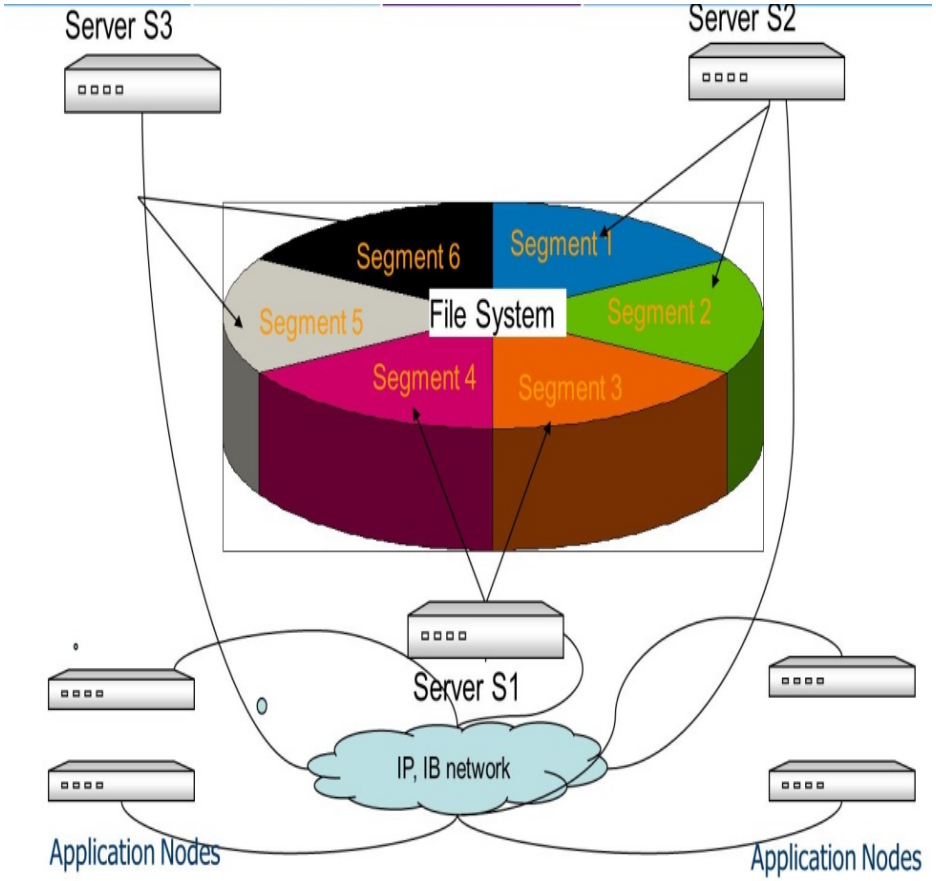
Hewlett-Packard Co.

Objectives

- Expose fundamentals of highly distributed segmented parallel file system architecture
- Review the challenges and goals of implementing de-duplication
- Show key points of the design:
  - Role of ES in de-duplication
  - Segmented Indexing and Index Segment Servers
  - Data Chunk Files, Manifests, Index Files
  - Representative Keys, Key Groups
- Questions

How to scale high?

Scalable segmented FS architecture



- Splits physical and logical space into segments
- Assigns control over storage segments to segment servers
- Segment servers are entirely responsible for file (inode) and block allocation **within the boundaries of the individual segments**
- segment servers coordinate caches and activities associated with objects **they maintain**
- Files and directories are distributed through the sets of segments

De-duplication Design: Goals

- **Efficiency** - design and implementation should be efficient enough to avoid degrading throughput of sequential write and read operations
- **Scalability** – the proposed design should be as scalable as the overall file system. In other words, adding more servers and segments to the de-duplication space should allow it to handle proportionally larger cumulative load
- **Manageability** – set of tools and utilities should be provided to de-duplicate, restore, copy, compare, replicate de-duplicated data, etc.

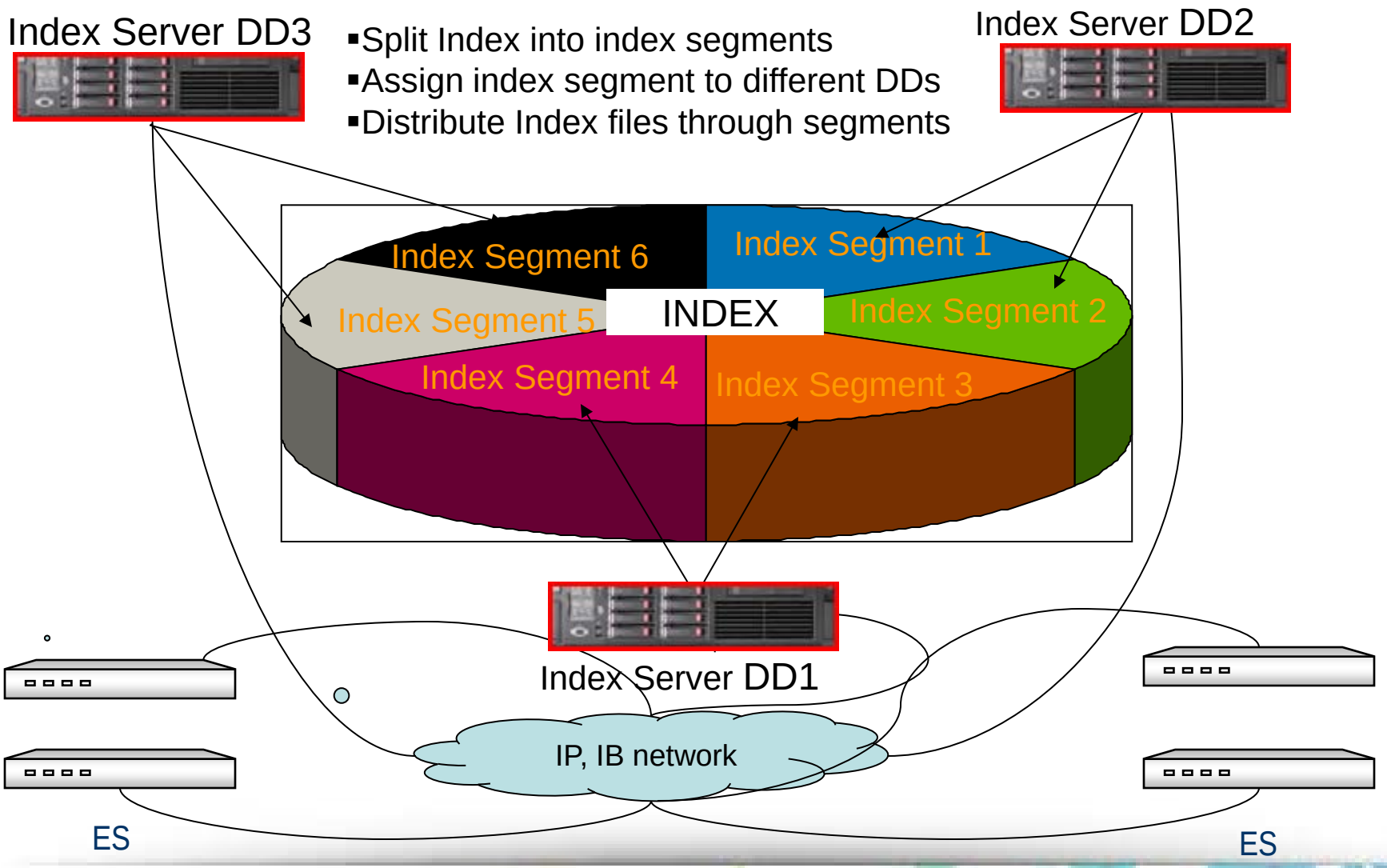
De-duplication Design Goals: Efficiency

- Avoid additional data hops
- ES determines location of data and reads it directly from corresponding DS or even directly from LUNs
- When writing, ES should send data directly to the DS for corresponding chunk store files (Dchunk file)
- Provide pre-allocation of contiguous spaces for data streams

De-duplication Design Goals: Scalability

- Distribute Dchunk files through Ibrix segments
- Multiple Dchunk files should accept new data at the same time
- Distribute indexes through multiple servers, so each server can support and cache only assigned part of the overall index tree

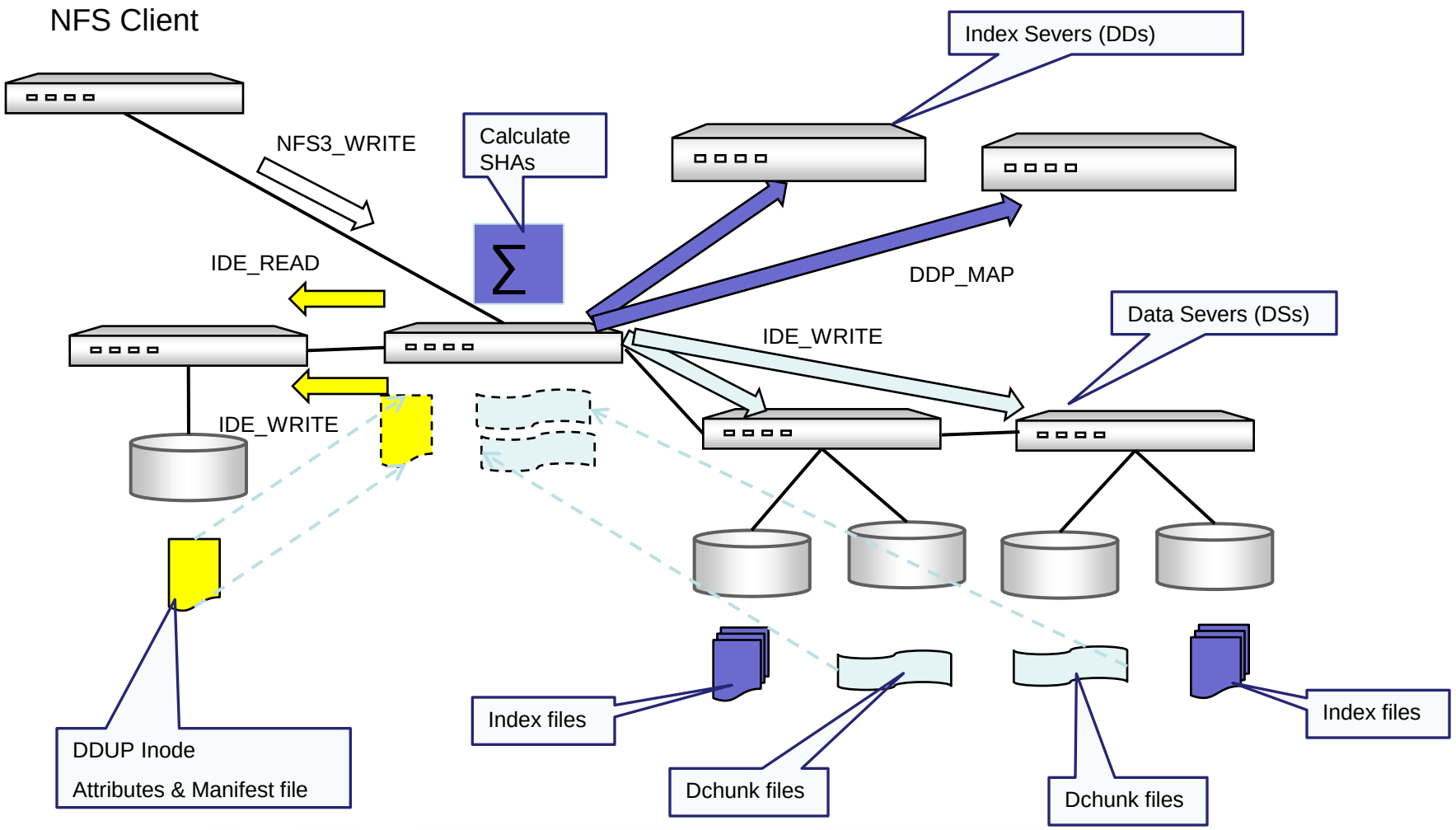
Segmented Indexing: DDup Index Servers (DD) and Index Segments



Components: DDup Servers (DD)

- Maintain assigned parts of overall Index (one or more index segments)
  - In-core representation of the index segments
  - On-disk representation of the index segments
- Releases unreferenced blocks of DChunk files
- Index segments may be reassigned from one DD to another any time as part of failover or for load balancing

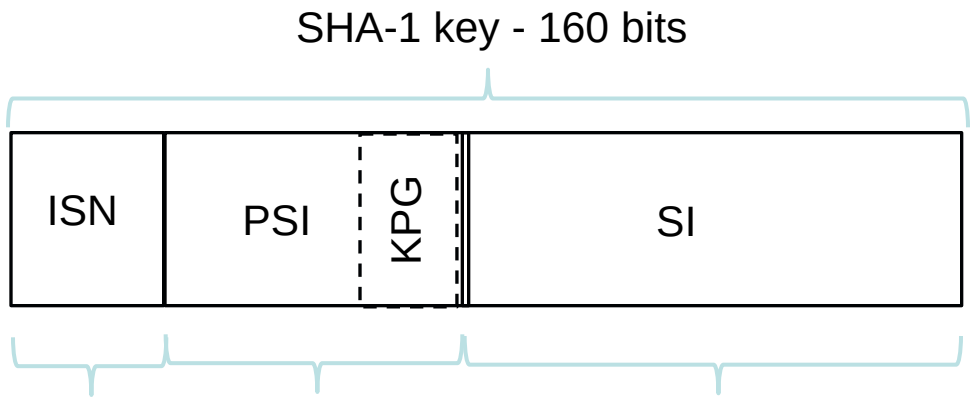
DDUP Dataflow – Write request



Segmented Indexing: Index composition

- 20 byte (160 bit) SHA-1 key is calculated by ES for any Addressable Data Block (ADB ~ 4K, 8K, etc.)
- This key is the key into Ibrix DDUP Index space
- DDUP Index space is divided into DDUP Index Segments
- These Index Segments can be handled by multiple Ibrix DD servers
- DDUP Index Segment Number (ISN) is a part of SHA key
- Association between segments and servers is known to every Ibrix ES; though it can change over time, is reasonably stable

Segmented Indexing: Index composition



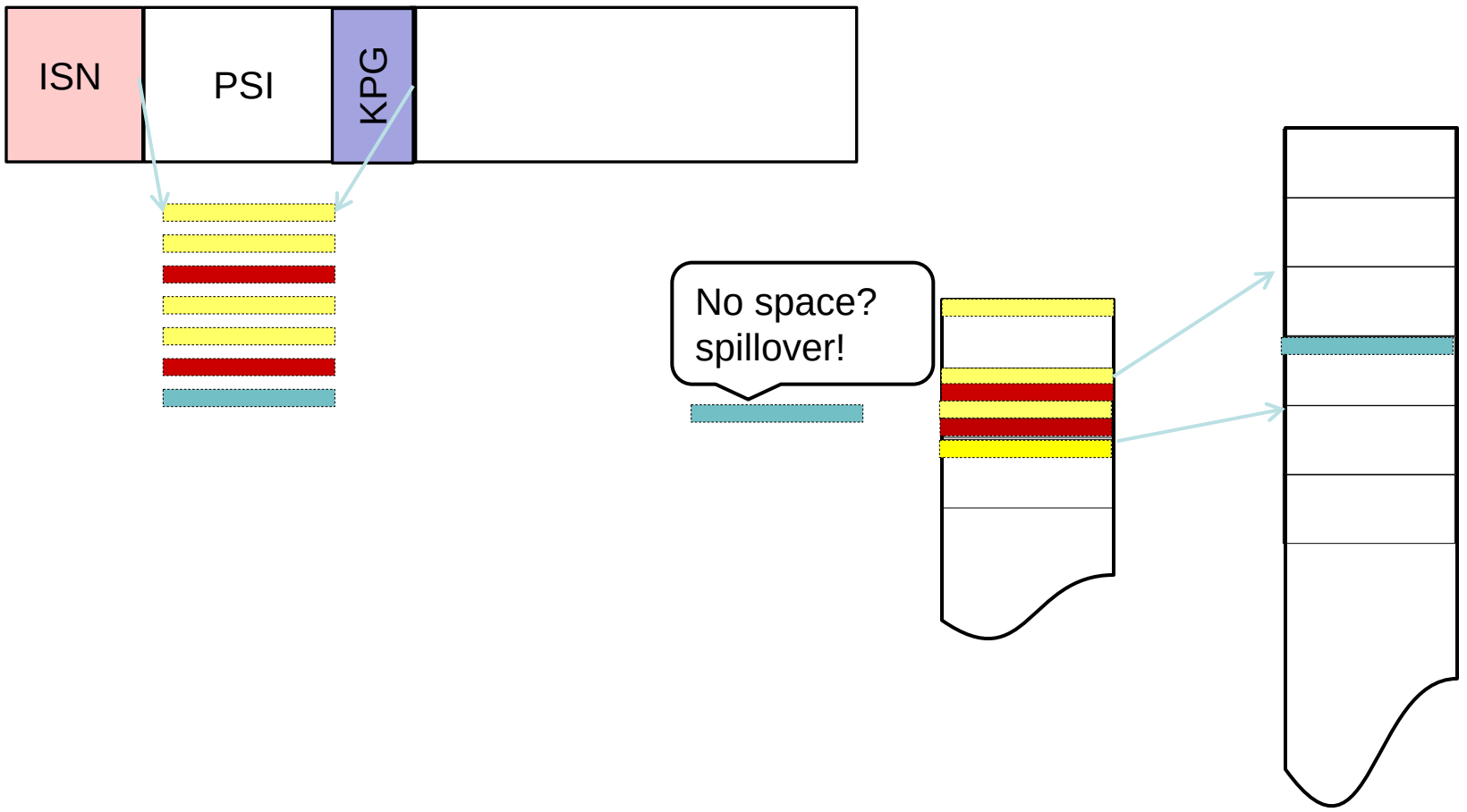
- The composition of ISN and PSI should be sufficient to cover the addressable data blocks (ADB) in the DDUP space.
- Example: if DDUP space is **512 TB** and the size of ADB is 4K, the number of the addressable element is 2^49 / 2^12 = 2^37. This can be divided into 256 (2^8) ISNs and 512M (2^29) PSIs.
- KPG – Key Placement Grouping of PSIs: all entries with the same PSI keys with zeroed KPG bits are placed into one storage group.

point to

DDup Servers: Index Files

- Direct Access files covering the space of the segment index
- Part of PSI is used as a record number
- Use key grouping to achieve better space utilization
- Spillover into the next level segment index file when run out of space in the group

DDup Servers: Index Files – KPG & spillover

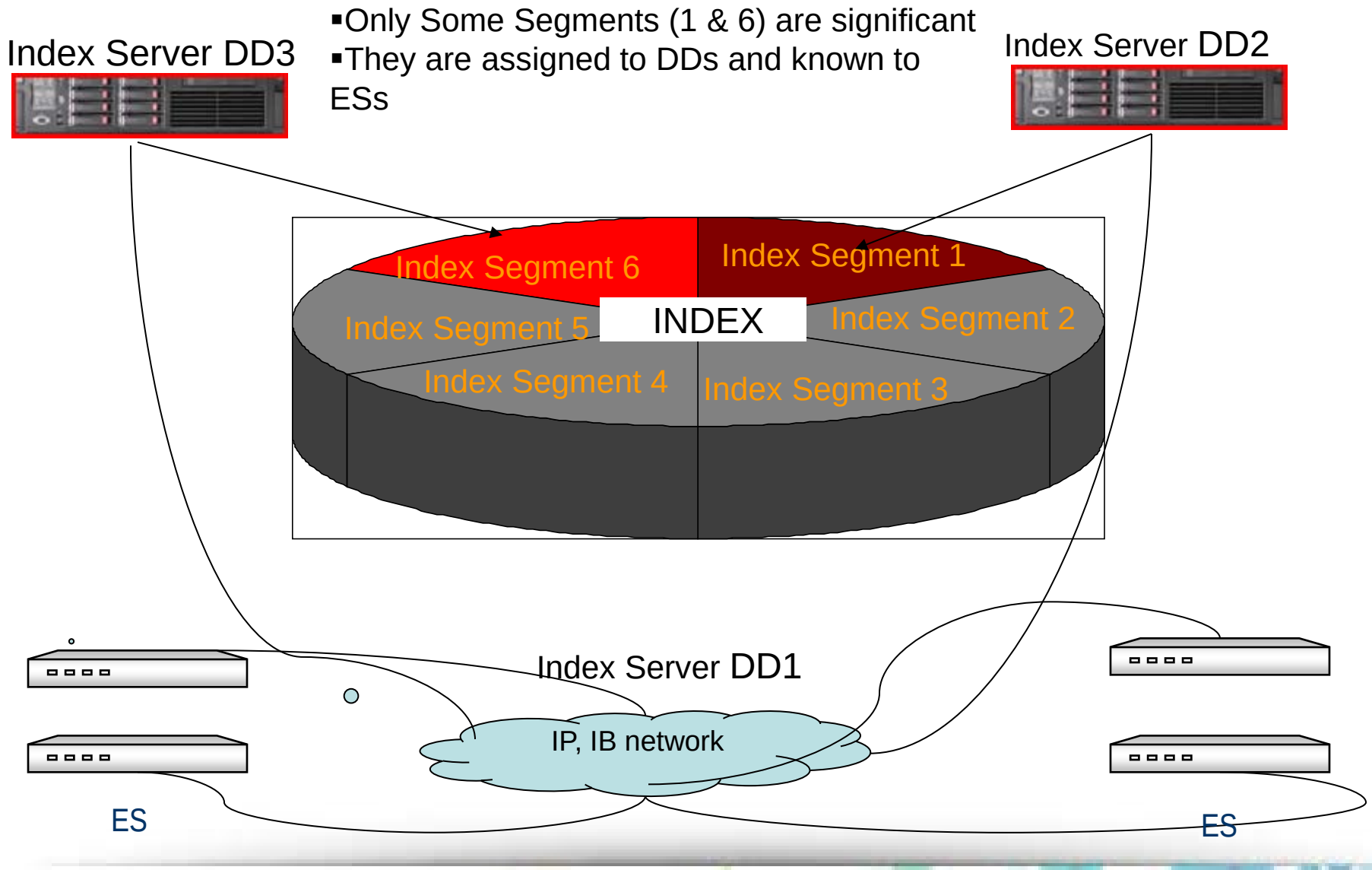


DDup Servers: Representative Indexing

- Use representative indexing to reduce in-core memory consumption
- Define ‘significant’ or representative index segments
- Only representative segments need to be assigned to Index Servers
- ES aggregates multiple keys into one DDP_MAP request starting with a representative key
- DD matches set of keys based on the value of the representative key; other keys are treated as associated
- DD keeps track of all data locations for all with representative and associated keys

DDup Servers: Representative Indexing

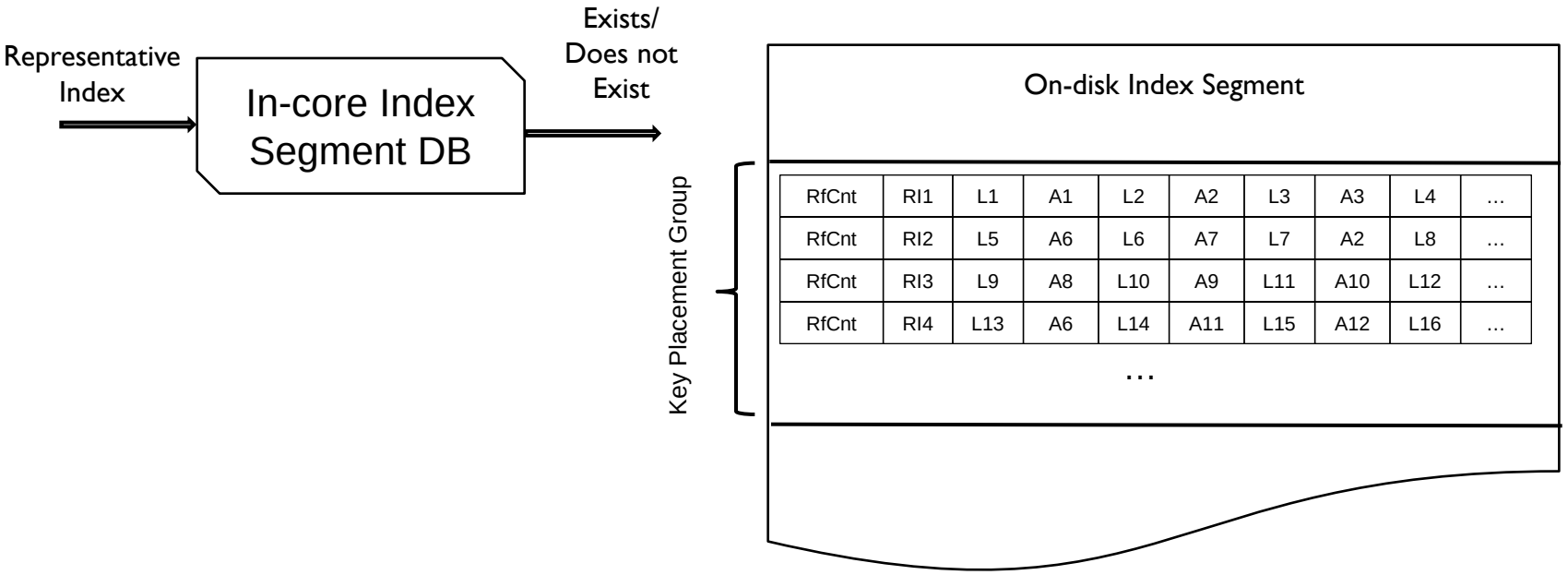
- Only Some Segments (1 & 6) are significant
- They are assigned to DDs and known to ESs



DDup Servers: Representative Indexing

DDB_MAP request includes one Representatives and several Associated Indexes and their Proposed locations

N	Representative Index	Proposed Location	Associated Index 1	Proposed Location	Associated Index 2	Proposed Location	…



Write performance implications

- We start writing into Dchunk files concurrently with sending DDB_MAP request
- DD should respond promptly if key is not found and writes should be committed
- When ES commits write it sends the DDB_COMMIT message to DD and data becomes available to other nodes
- If key exists DD fetches the record from the Index file, bumps up the RefCount and replies with the collation of the representative and associated key; in this case writes to Dchunk files are aborted
- Most of the actions above are not on IO path and do not slow down writing

Dchunks & contiguous space allocation

- Multiple Dchunk files are active at the same time
- ES pre-allocates space in Dchunk files and maintains affinity between incoming data streams and chunks
- Various block distribution policies can be applied. For example we can implement striping this way
- DD servers record the key to location mappings and maintain reference counts
- Special allocation policies may be added to place Dchunk files on designated set of segments
- Dchunk files can be cloned
- Dchunk files can be compressed and encrypted

DDup inodes: manifests & properties

- Various types of de-duplicated files are recognized by Ibrix FS and various formats of manifests can be supported
- Special allocation policies are used to place de-duplicated inodes on designated set of segments
- Regular file attributes are maintained on the inode level

Opportunistic De-duplication

Goal: remove the de-duplication process itself from the write performance path and prevent potentially slower mechanism of index lookup to negatively affect efficiency of writes.

- ❑ DD may respond with the instruction to write some of the data chunks as **not de-duplicated**
- ❑ ES may decide also not to de-duplicate some or all the chunks
  - ✓heuristic analyses of responsiveness of DD requests and responsiveness of Dchunk DS
  - ✓ratio of newly seen to already know data.

Questions?