Effiziente In-Memory Verarbeitung von SPARQL-Anfragen auf großen Datenmengen

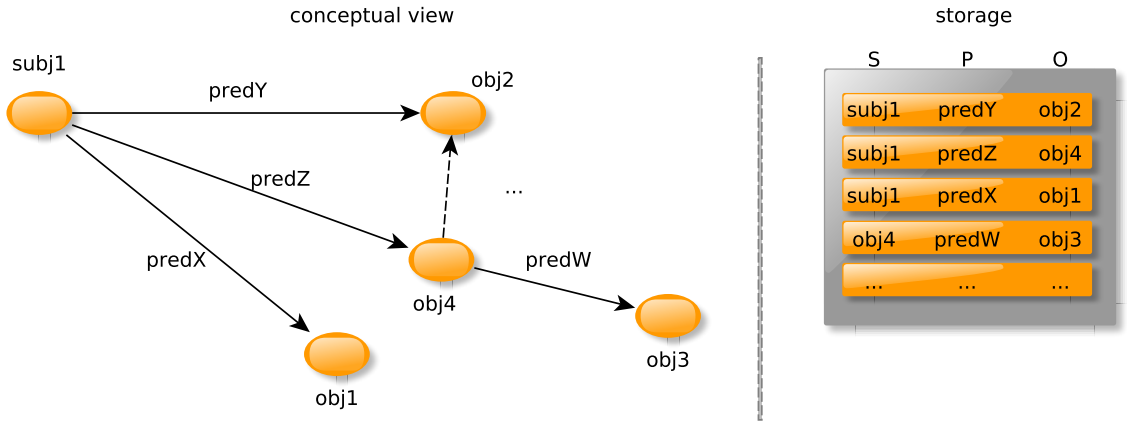
–Verteidigungsvortrag v0.$\overline{9}$–
Masterarbeit

Steve Göring

18. Dezember 2013

Gliederung

Intro



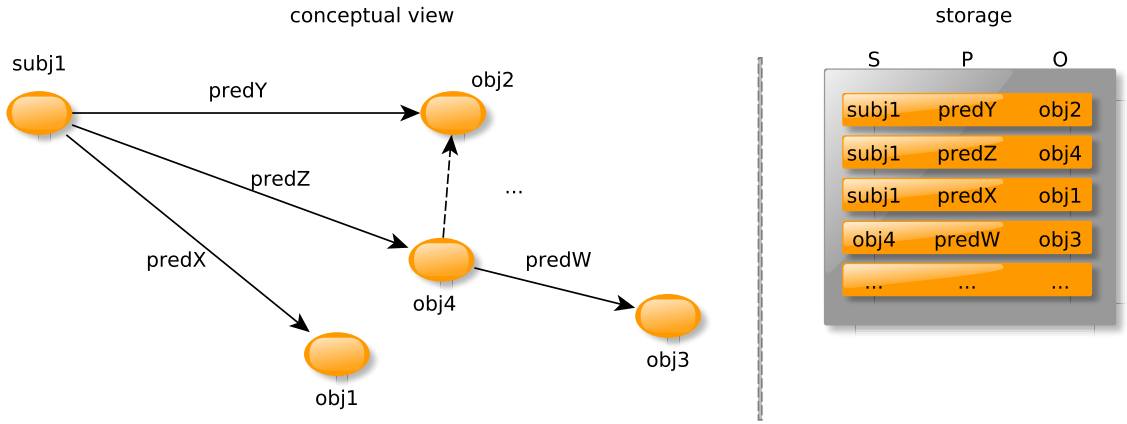
S $< http://dbpedia.org/resource/Star\_Trek : \_Enterprise >$

P $< http://dbpedia.org/ontology/abstract >$

O “Star Trek: Enterprise (originally titled simply Enterprise....“

- Linked-Open-Data $\rightarrow$ RDF-Tripel $\rightarrow$ SPO-Aussagen
- moderne Rechnerarchitekturen
- In-Memory
- relationale DBs $\rightarrow$ BESSER: spezielle DBs für LOD

Intro



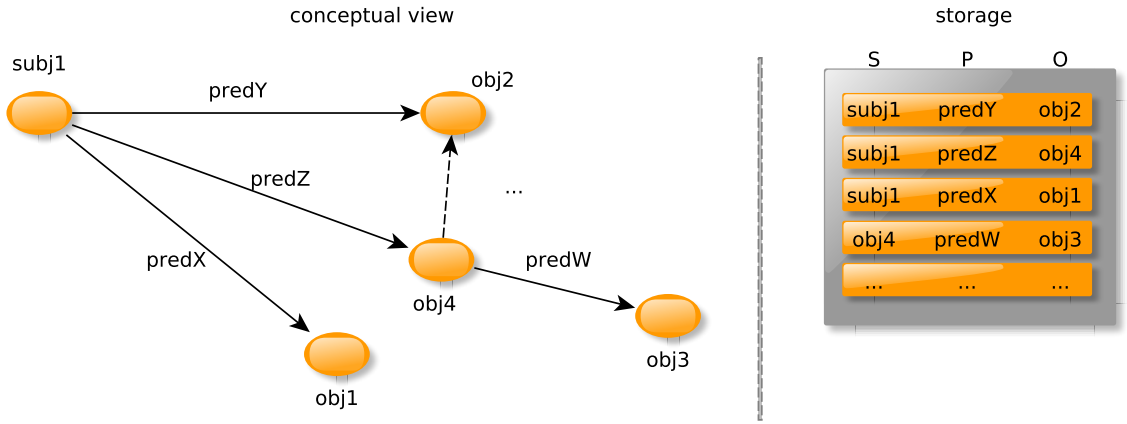
S $< http://dbpedia.org/resource/Star\_Trek : \_Enterprise >$

P $< http://dbpedia.org/ontology/abstract >$

O “Star Trek: Enterprise (originally titled simply Enterprise....“

- Linked-Open-Data $\rightarrow$ RDF-Tripel $\rightarrow$ SPO-Aussagen
- moderne Rechnerarchitekturen
- In-Memory
- relationale DBs $\rightarrow$ BESSER: spezielle DBs für LOD

Intro



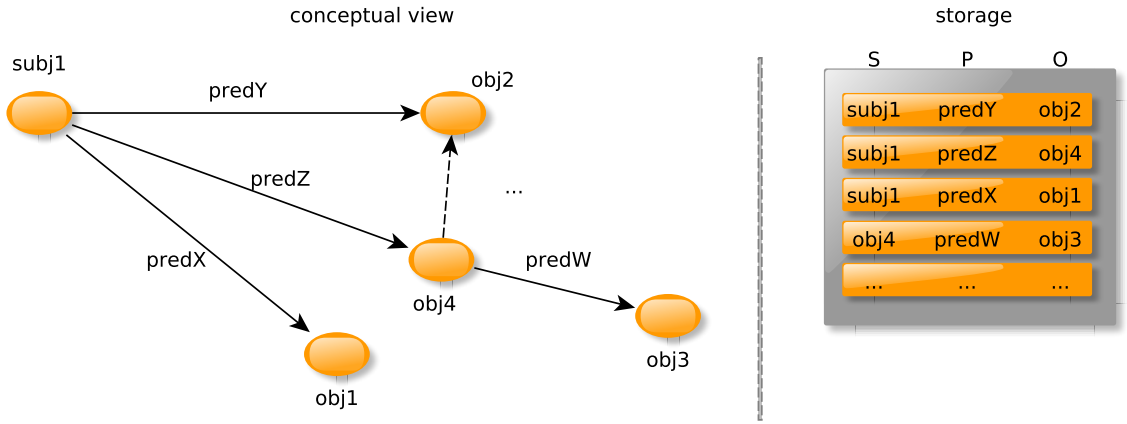
S $< http://dbpedia.org/resource/Star\_Trek:\_Enterprise >$

P $< http://dbpedia.org/ontology/abstract >$

O “Star Trek: Enterprise (originally titled simply Enterprise....“

- Linked-Open-Data $\rightarrow$ RDF-Tripel $\rightarrow$ SPO-Aussagen
- moderne Rechnerarchitekturen
- In-Memory
- relationale DBs $\rightarrow$ BESSER: spezielle DBs für LOD

Intro



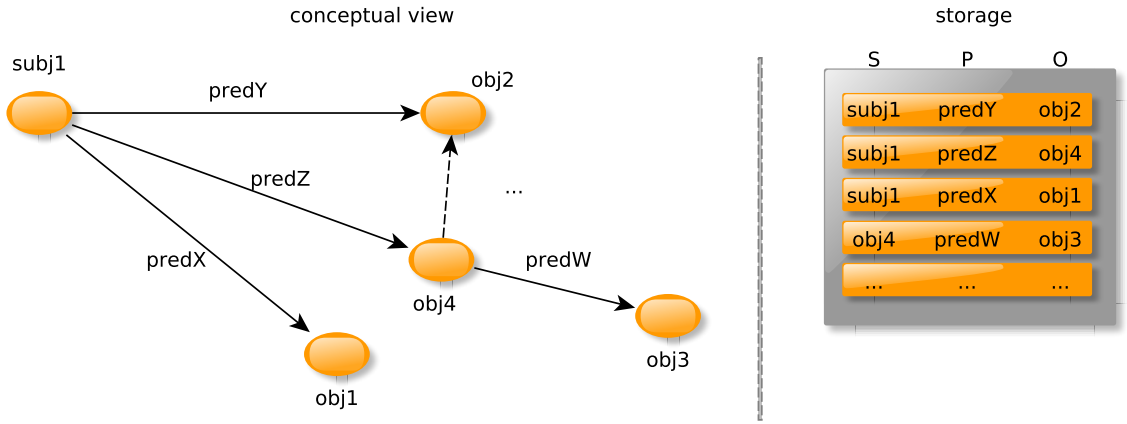
S $< http://dbpedia.org/resource/Star\_Trek : \_Enterprise >$

P $< http://dbpedia.org/ontology/abstract >$

O “Star Trek: Enterprise (originally titled simply Enterprise....“

- Linked-Open-Data $\rightarrow$ RDF-Tripel $\rightarrow$ SPO-Aussagen
- moderne Rechnerarchitekturen
- In-Memory
- relationale DBs $\rightarrow$ BESSER: spezielle DBs für LOD

Intro



S $< http://dbpedia.org/resource/Star\_Trek : \_Enterprise >$

P $< http://dbpedia.org/ontology/abstract >$

O “Star Trek: Enterprise (originally titled simply Enterprise....“

- Linked-Open-Data $\rightarrow$ RDF-Tripel $\rightarrow$ SPO-Aussagen
- moderne Rechnerarchitekturen
- In-Memory
- relationale DBs $\rightarrow$ BESSER: spezielle DBs für LOD

Ziele

- bestehender Prototyp (CameLOD)
- Steigerung der Effizienz (Laufzeit/Speicher)
- Ausnutzung moderner CPU-Features (Analyse+ AutoVec)
- genaue Analyse & Profiling des bestehenden Systems
- $\rightarrow$ Konzepte/Mikrobenchs $\rightarrow$ Verbesserungen
- Evaluierung mittels Vergleich vorher vs. nachher

Ziele

- bestehender Prototyp (CameLOD)
- Steigerung der Effizienz (Laufzeit/Speicher)
- Ausnutzung moderner CPU-Features (Analyse+ AutoVec)
- genaue Analyse & Profiling des bestehenden Systems
- $\rightarrow$ Konzepte/Mikrobenchs $\rightarrow$ Verbesserungen
- Evaluierung mittels Vergleich vorher vs. nachher

Ziele

- bestehender Prototyp (CameLOD)
- Steigerung der Effizienz (Laufzeit/Speicher)
- Ausnutzung moderner CPU-Features (Analyse+ AutoVec)
- genaue Analyse & Profiling des bestehenden Systems
- $\rightarrow$ Konzepte/Mikrobenchs $\rightarrow$ Verbesserungen
- Evaluierung mittels Vergleich vorher vs. nachher

Ziele

- bestehender Prototyp (CameLOD)
- Steigerung der Effizienz (Laufzeit/Speicher)
- Ausnutzung moderner CPU-Features (Analyse+ AutoVec)
- genaue Analyse & Profiling des bestehenden Systems
- $\rightarrow$ Konzepte/Mikrobenchs $\rightarrow$ Verbesserungen
- Evaluierung mittels Vergleich vorher vs. nachher

Ziele

- bestehender Prototyp (CameLOD)
- Steigerung der Effizienz (Laufzeit/Speicher)
- Ausnutzung moderner CPU-Features (Analyse+ AutoVec)
- genaue Analyse & Profiling des bestehenden Systems
- $\rightarrow$ Konzepte/Mikrobenchs $\rightarrow$ Verbesserungen
- Evaluierung mittels Vergleich vorher vs. nachher

Ziele

- bestehender Prototyp (CameLOD)
- Steigerung der Effizienz (Laufzeit/Speicher)
- Ausnutzung moderner CPU-Features (Analyse+ AutoVec)
- genaue Analyse & Profiling des bestehenden Systems
- $\rightarrow$ Konzepte/Mikrobenchs $\rightarrow$ Verbesserungen
- Evaluierung mittels Vergleich vorher vs. nachher

Ziele

- bestehender Prototyp (CameLOD)
- Steigerung der Effizienz (Laufzeit/Speicher)
- Ausnutzung moderner CPU-Features (Analyse+ AutoVec)
- genaue Analyse & Profiling des bestehenden Systems
- $\rightarrow$ Konzepte/Mikrobenchs $\rightarrow$ Verbesserungen
- Evaluierung mittels Vergleich vorher vs. nachher

CameLOD

- In-Memory DB für LOD
- index-all
- Wörterbuchkompression
- Verarbeitung auf Integer Ebene
- Datengrundlage: DBPedia

CameLOD

- In-Memory DB für LOD
- index-all
- Wörterbuchkompression
- Verarbeitung auf Integer Ebene
- Datengrundlage: DBPedia

CameLOD

- In-Memory DB für LOD
- index-all
- Wörterbuchkompression
- Verarbeitung auf Integer Ebene
- Datengrundlage: DBPedia

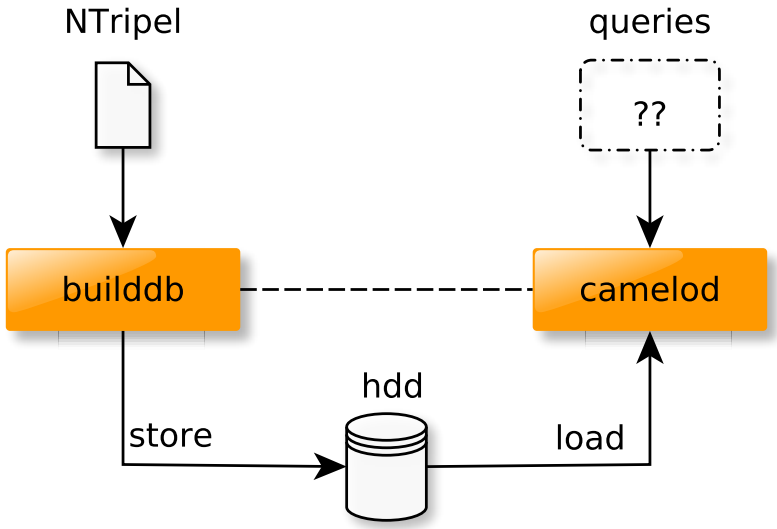
CameLOD

- In-Memory DB für LOD
- index-all
- Wörterbuchkompression
- Verarbeitung auf Integer Ebene
- Datengrundlage: DBPedia

CameLOD

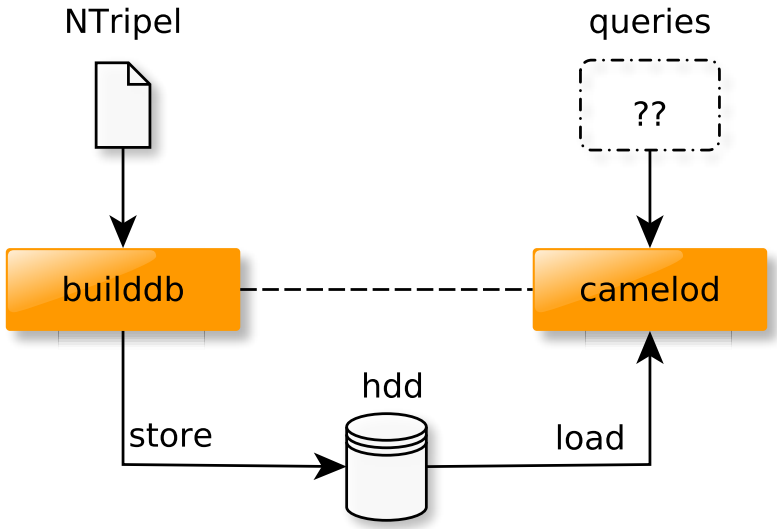
- In-Memory DB für LOD
- index-all
- Wörterbuchkompression
- Verarbeitung auf Integer Ebene
- Datengrundlage: DBPedia

Ablauf



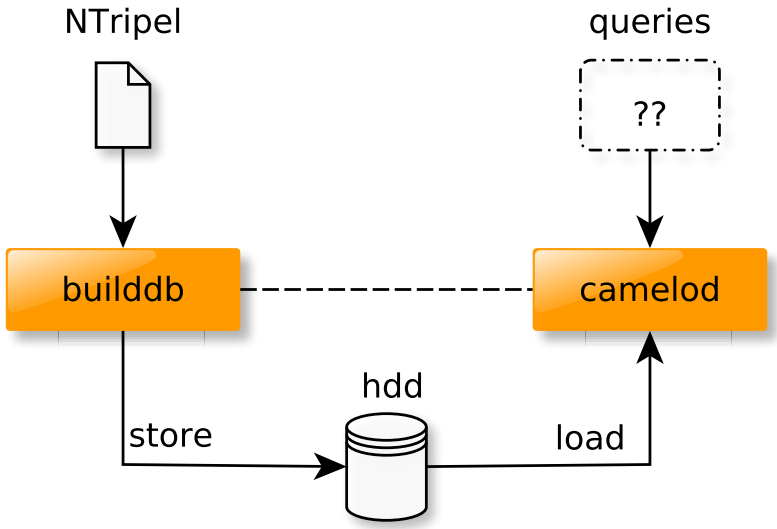
- Aufbau der Datenbank (Kompression)
- → serialisierte permanente Kopie auf HDD
- Laden von HDD + Anfragen

Ablauf



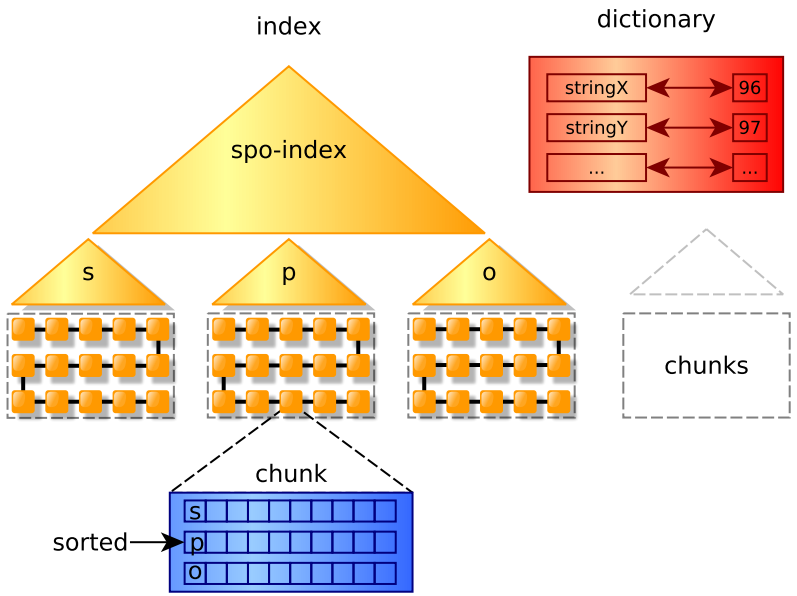
- Aufbau der Datenbank (Kompression)
- $\rightarrow$ serialisierte permanente Kopie auf HDD
- Laden von HDD + Anfragen

Ablauf



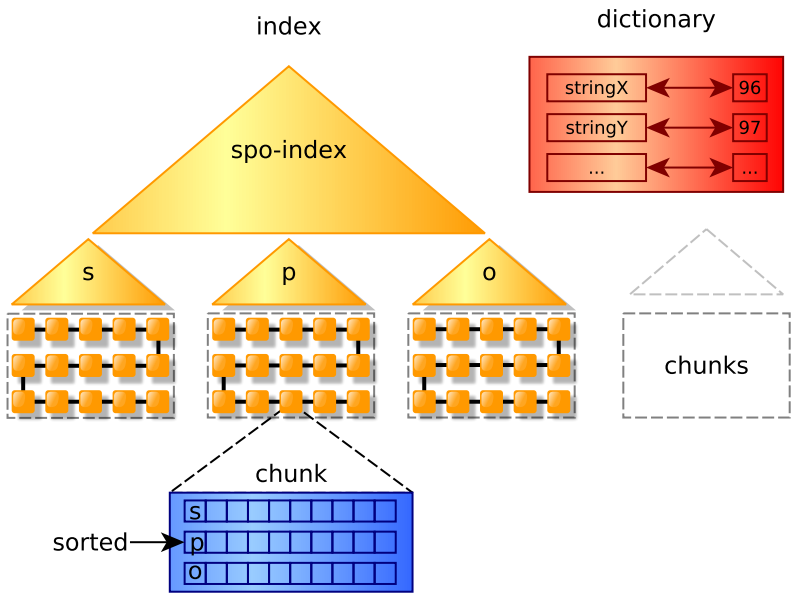
- Aufbau der Datenbank (Kompression)
- $\rightarrow$ serialisierte permanente Kopie auf HDD
- Laden von HDD + Anfragen

Struktur



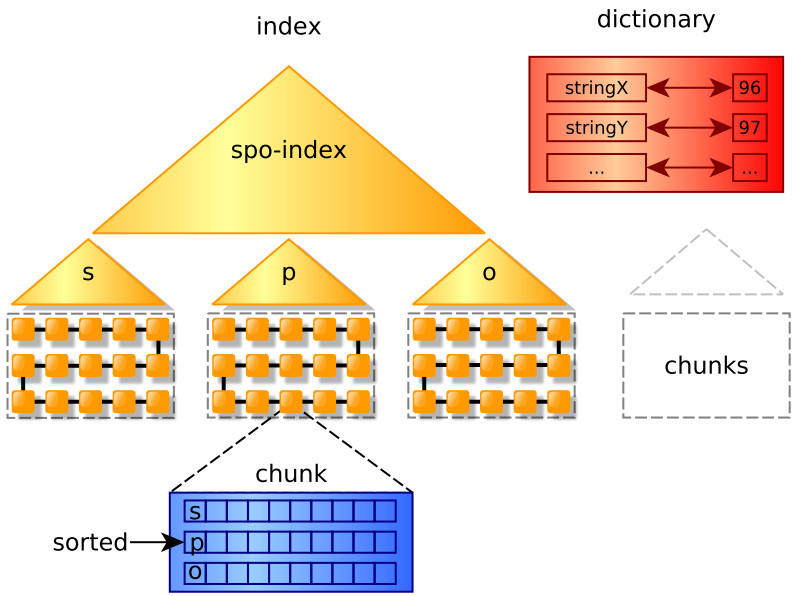
- Index für S,P,O
- Wörterbuch (sortierte Zuordnung)
- Anfragen auf Index-Ebene

Struktur



- Index für S,P,O
- Wörterbuch (sortierte Zuordnung)
- Anfragen auf Index-Ebene

Struktur



- Index für S,P,O
- Wörterbuch (sortierte Zuordnung)
- Anfragen auf Index-Ebene

Profiling

- mit VTune
- builddb
- camelod: 13 Testanfragen (join, parallelisiert, scan, filter)
  - Q1, Q2: Scan + Aggregation + Filterung
  - Q3: index-join + Filterung + paralleler Scan
  - Q4: Scan + Aggregation + Sortierung
  - Q5: Scan + Filterung + StarJoin
- Kompression
- allgemeine Verbesserungen

Profiling

- mit VTune
- builddb
- camelod: 13 Testanfragen (join, parallelisiert, scan, filter)
  - Q1, Q2: Scan + Aggregation + Filterung
  - Q3: index-join + Filterung + paralleler Scan
  - Q4: Scan + Aggregation + Sortierung
  - Q5: Scan + Filterung + StarJoin
- Kompression
- allgemeine Verbesserungen

Profiling

- mit VTune
- builddb
- camelod: 13 Testanfragen (join, parallelisiert, scan, filter)
  - Q1, Q2: Scan + Aggregation + Filterung
  - Q3: index-join + Filterung + paralleler Scan
  - Q4: Scan + Aggregation + Sortierung
  - Q5: Scan + Filterung + StarJoin
- Kompression
- allgemeine Verbesserungen

Profiling

- mit VTune
- builddb
- camelod: 13 Testanfragen (join, parallelisiert, scan, filter)
  - Q1, Q2: Scan + Aggregation + Filterung
  - Q3: index-join + Filterung + paralleler Scan
  - Q4: Scan + Aggregation + Sortierung
  - Q5: Scan + Filterung + StarJoin
- Kompression
- allgemeine Verbesserungen

Profiling

- mit VTune
- builddb
- camelod: 13 Testanfragen (join, parallelisiert, scan, filter)
  - Q1, Q2: Scan + Aggregation + Filterung
  - Q3: index-join + Filterung + paralleler Scan
  - Q4: Scan + Aggregation + Sortierung
  - Q5: Scan + Filterung + StarJoin
- Kompression
- allgemeine Verbesserungen

Profiling

- mit VTune
- builddb
- camelod: 13 Testanfragen (join, parallelisiert, scan, filter)
  - Q1, Q2: Scan + Aggregation + Filterung
  - Q3: index-join + Filterung + paralleler Scan
  - Q4: Scan + Aggregation + Sortierung
  - Q5: Scan + Filterung + StarJoin
- Kompression
- allgemeine Verbesserungen

Profiling

- mit VTune
- builddb
- camelod: 13 Testanfragen (join, parallelisiert, scan, filter)
  - Q1, Q2: Scan + Aggregation + Filterung
  - Q3: index-join + Filterung + paralleler Scan
  - Q4: Scan + Aggregation + Sortierung
  - Q5: Scan + Filterung + StarJoin
- Kompression
- allgemeine Verbesserungen

Profiling - builddb

	Hash-	Kollisionen	Chunks	Find & insert	Smart-Pointer
Builddb	$\approx 1\%$	$\approx 4\%$	$\approx 12\%$		$\approx 42\%$

- $\approx 2\ h$
- Auslastung eines Kerns
- Find&Insert: sequentiell $\rightarrow$ binäre Suche
- SmartPointer

Profiling - CameLOD

	BitVektoren	Smart-Pointer
Q1	$\approx 1.8\%$	$\approx 35\%$
Q2	$\approx 1\%$	$\approx 40\%$
Q3	$\approx 69\%$	$\approx 0.5\%$
Q4	$\approx 1\%$	$\approx 36\%$
Q5	$\approx 56\%$	$\approx 3.5\%$

- parallelisierte Anfragen: gut
- Join: viele BitVec Operationen
- SmartPointer

Profiling - Kompression + allgemein

- DBPedia $\approx$ 30 *GB* $\rightarrow$ 18 *GB*
- $\approx$ 4.8 *GB* pro Index $\rightarrow$ 14.4 *GB*
- Chunk-Vektoren: 64 *Bit* uLong Werte
  - viele Blöcke
  - Chunk-Größe: 2000 $\rightarrow$ keine Ausnutzung von uLong
- Dictionary: viele Präfixe

allgemein:

- paralleles Laden/Speichern der Indizes
- binäre Suche

Profiling - Kompression + allgemein

- DBPedia $\approx$ 30 *GB* $\rightarrow$ 18 *GB*
- $\approx$ 4.8 *GB* pro Index $\rightarrow$ 14.4 *GB*
- Chunk-Vektoren: 64 *Bit* uLong Werte
  - viele Blöcke
  - Chunk-Größe: 2000 $\rightarrow$ keine Ausnutzung von uLong
- Dictionary: viele Präfixe

allgemein:

- paralleles Laden/Speichern der Indizes
- binäre Suche

Profiling - Kompression + allgemein

- DBPedia $\approx$ 30 *GB* $\rightarrow$ 18 *GB*
- $\approx$ 4.8 *GB* pro Index $\rightarrow$ 14.4 *GB*
- Chunk-Vektoren: 64 *Bit* uLong Werte
  - viele Blöcke
  - Chunk-Größe: 2000 $\rightarrow$ keine Ausnutzung von uLong
- Dictionary: viele Präfixe

allgemein:

- paralleles Laden/Speichern der Indizes
- binäre Suche

Profiling - Kompression + allgemein

- DBPedia $\approx$ 30 *GB* $\rightarrow$ 18 *GB*
- $\approx$ 4.8 *GB* pro Index $\rightarrow$ 14.4 *GB*
- Chunk-Vektoren: 64 *Bit* uLong Werte
  - viele Blöcke
  - Chunk-Größe: 2000 $\rightarrow$ keine Ausnutzung von uLong
- Dictionary: viele Präfixe

allgemein:

- paralleles Laden/Speichern der Indizes
- binäre Suche

Profiling - Kompression + allgemein

- DBPedia $\approx$ 30 *GB* $\rightarrow$ 18 *GB*
- $\approx$ 4.8 *GB* pro Index $\rightarrow$ 14.4 *GB*
- Chunk-Vektoren: 64 *Bit* uLong Werte
  - viele Blöcke
  - Chunk-Größe: 2000 $\rightarrow$ keine Ausnutzung von uLong
- Dictionary: viele Präfixe

allgemein:

- paralleles Laden/Speichern der Indizes
- binäre Suche

Profiling - Kompression + allgemein

- DBPedia $\approx$ 30 *GB* $\rightarrow$ 18 *GB*
- $\approx$ 4.8 *GB* pro Index $\rightarrow$ 14.4 *GB*
- Chunk-Vektoren: 64 *Bit* uLong Werte
  - viele Blöcke
  - Chunk-Größe: 2000 $\rightarrow$ keine Ausnutzung von uLong
- Dictionary: viele Präfixe

allgemein:

- paralleles Laden/Speichern der Indizes
- binäre Suche

Profiling - Kompression + allgemein

- DBPedia $\approx$ 30 *GB* $\rightarrow$ 18 *GB*
- $\approx$ 4.8 *GB* pro Index $\rightarrow$ 14.4 *GB*
- Chunk-Vektoren: 64 *Bit* uLong Werte
  - viele Blöcke
  - Chunk-Größe: 2000 $\rightarrow$ keine Ausnutzung von uLong
- Dictionary: viele Präfixe

allgemein:

- paralleles Laden/Speichern der Indizes
- binäre Suche

Ansätze

- SmartPointer
- BitVektoren: $>$ 20% Zeitgewinn (Joins)
- Hash-Funktion: keine Kollisionen
- Kompression: FOR, RLE, Dict
  - FOR: $\approx$ 56% Speichergewinn
  - Dict: $\approx$ 30% Gewinn
- Speicher: NUMA, MMF
  - NUMA: schlecht
  - MMF: gut
- Chunk-Größe
  - guter Kompromiss: 1000 Elemente (2000 Max)

$\rightarrow$ SmartPointer + RLE-Kompression

Ansätze

- SmartPointer
- BitVektoren: $>$ 20% Zeitgewinn (Joins)
- Hash-Funktion: keine Kollisionen
- Kompression: FOR, RLE, Dict
  - FOR: $\approx$ 56% Speichergewinn
  - Dict: $\approx$ 30% Gewinn
- Speicher: NUMA, MMF
  - NUMA: schlecht
  - MMF: gut
- Chunk-Größe
  - guter Kompromiss: 1000 Elemente (2000 Max)

$\rightarrow$ SmartPointer + RLE-Kompression

Ansätze

- SmartPointer
- BitVektoren: $>$ 20% Zeitgewinn (Joins)
- Hash-Funktion: keine Kollisionen
- Kompression: FOR, RLE, Dict
  - FOR: $\approx$ 56% Speichergewinn
  - Dict: $\approx$ 30% Gewinn
- Speicher: NUMA, MMF
  - NUMA: schlecht
  - MMF: gut
- Chunk-Größe
  - guter Kompromiss: 1000 Elemente (2000 Max)

$\rightarrow$ SmartPointer + RLE-Kompression

Ansätze

- SmartPointer
- BitVektoren: $>$ 20% Zeitgewinn (Joins)
- Hash-Funktion: keine Kollisionen
- Kompression: FOR, RLE, Dict
  - FOR: $\approx$ 56% Speichergewinn
  - Dict: $\approx$ 30% Gewinn
- Speicher: NUMA, MMF
  - NUMA: schlecht
  - MMF: gut
- Chunk-Größe
  - guter Kompromiss: 1000 Elemente (2000 Max)

$\rightarrow$ SmartPointer + RLE-Kompression

Ansätze

- SmartPointer
- BitVektoren: $>$ 20% Zeitgewinn (Joins)
- Hash-Funktion: keine Kollisionen
- Kompression: FOR, RLE, Dict
  - FOR: $\approx$ 56% Speichergewinn
  - Dict: $\approx$ 30% Gewinn
- Speicher: NUMA, MMF
  - NUMA: schlecht
  - MMF: gut
- Chunk-Größe
  - guter Kompromiss: 1000 Elemente (2000 Max)

$\rightarrow$ SmartPointer + RLE-Kompression

Ansätze

- SmartPointer
- BitVektoren: $>$ 20% Zeitgewinn (Joins)
- Hash-Funktion: keine Kollisionen
- Kompression: FOR, RLE, Dict
  - FOR: $\approx$ 56% Speichergewinn
  - Dict: $\approx$ 30% Gewinn
- Speicher: NUMA, MMF
  - NUMA: schlecht
  - MMF: gut
- Chunk-Größe
  - guter Kompromiss: 1000 Elemente (2000 Max)

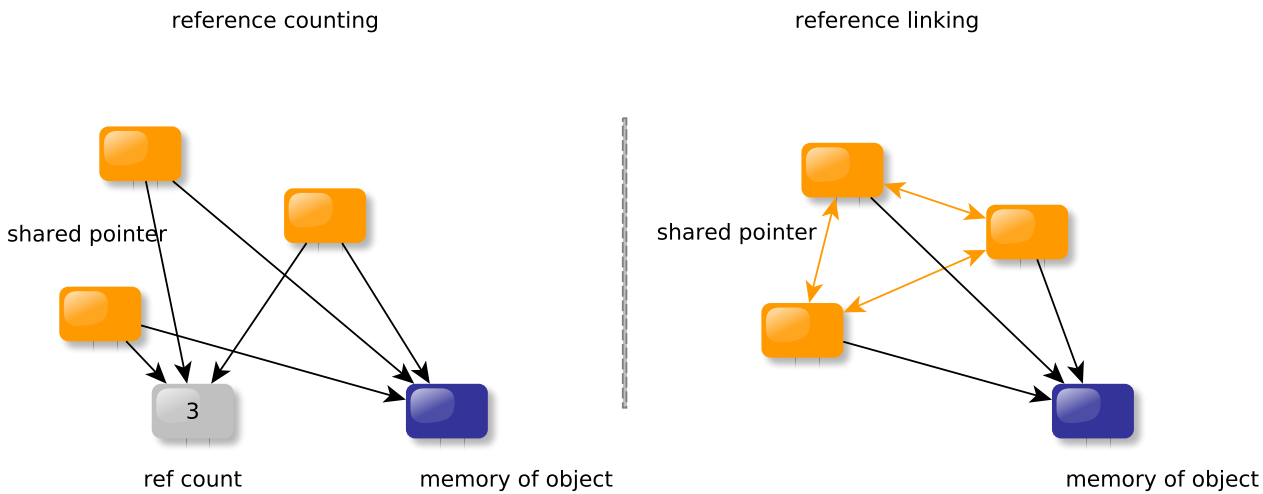
$\rightarrow$ SmartPointer + RLE-Kompression

Ansätze

- SmartPointer
- BitVektoren: $>$ 20% Zeitgewinn (Joins)
- Hash-Funktion: keine Kollisionen
- Kompression: FOR, RLE, Dict
  - FOR: $\approx$ 56% Speichergewinn
  - Dict: $\approx$ 30% Gewinn
- Speicher: NUMA, MMF
  - NUMA: schlecht
  - MMF: gut
- Chunk-Größe
  - guter Kompromiss: 1000 Elemente (2000 Max)

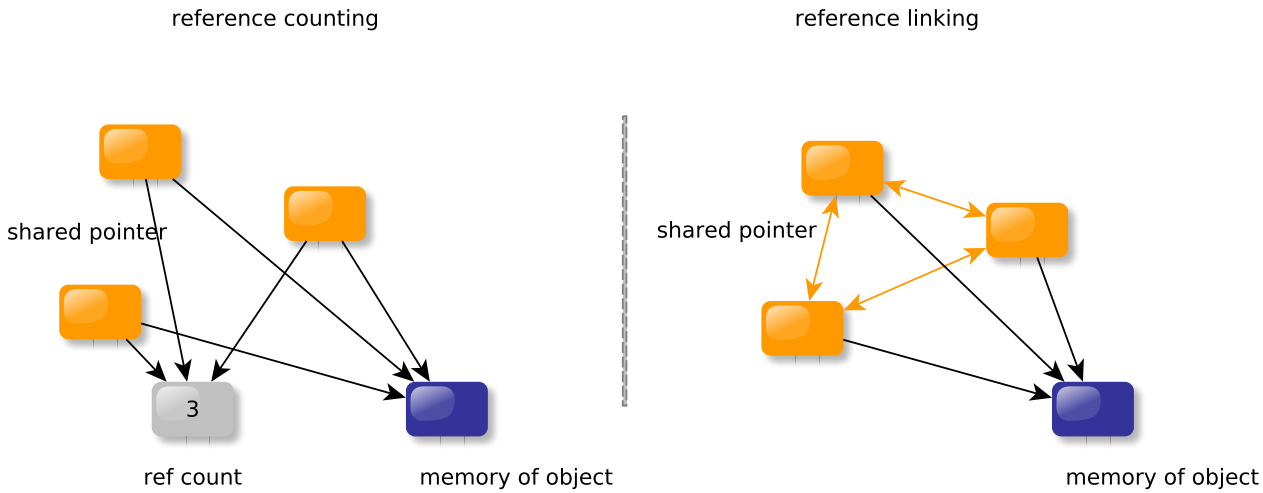
$\rightarrow$ SmartPointer + RLE-Kompression

Smart Pointer



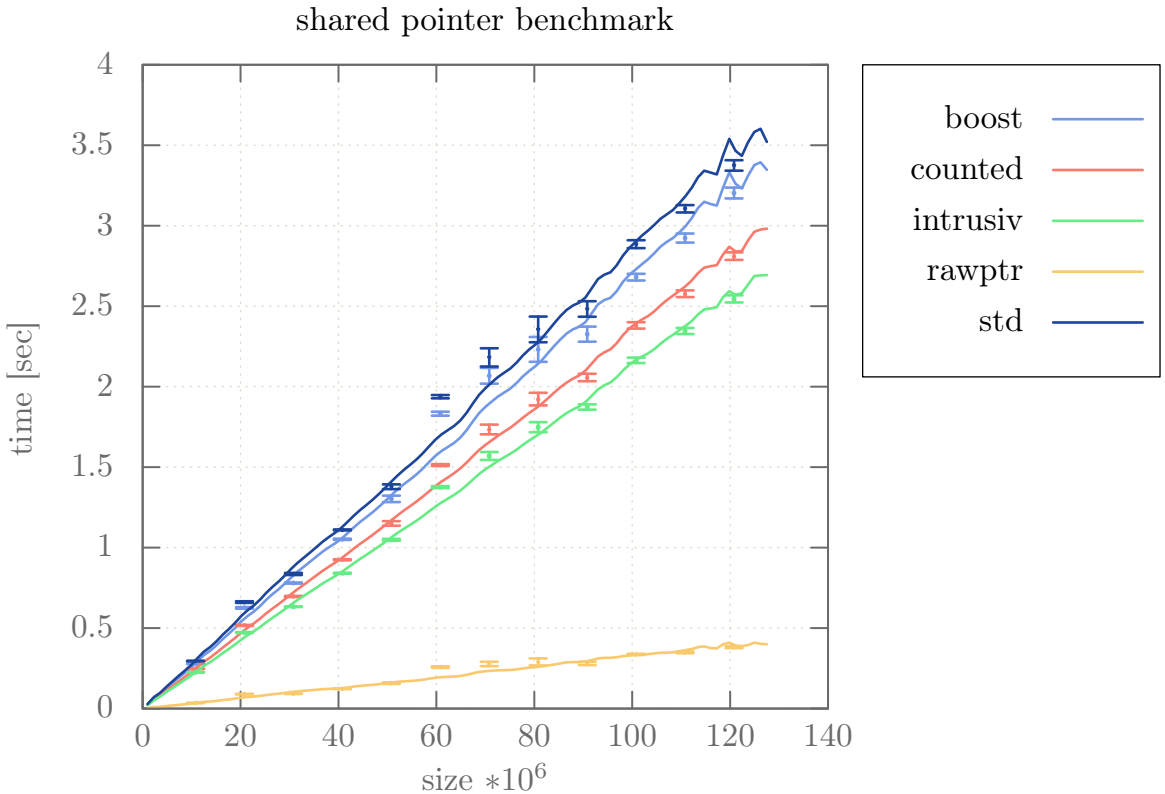
- Shared Pointer
- Referenzzählung → sync Counter

Smart Pointer



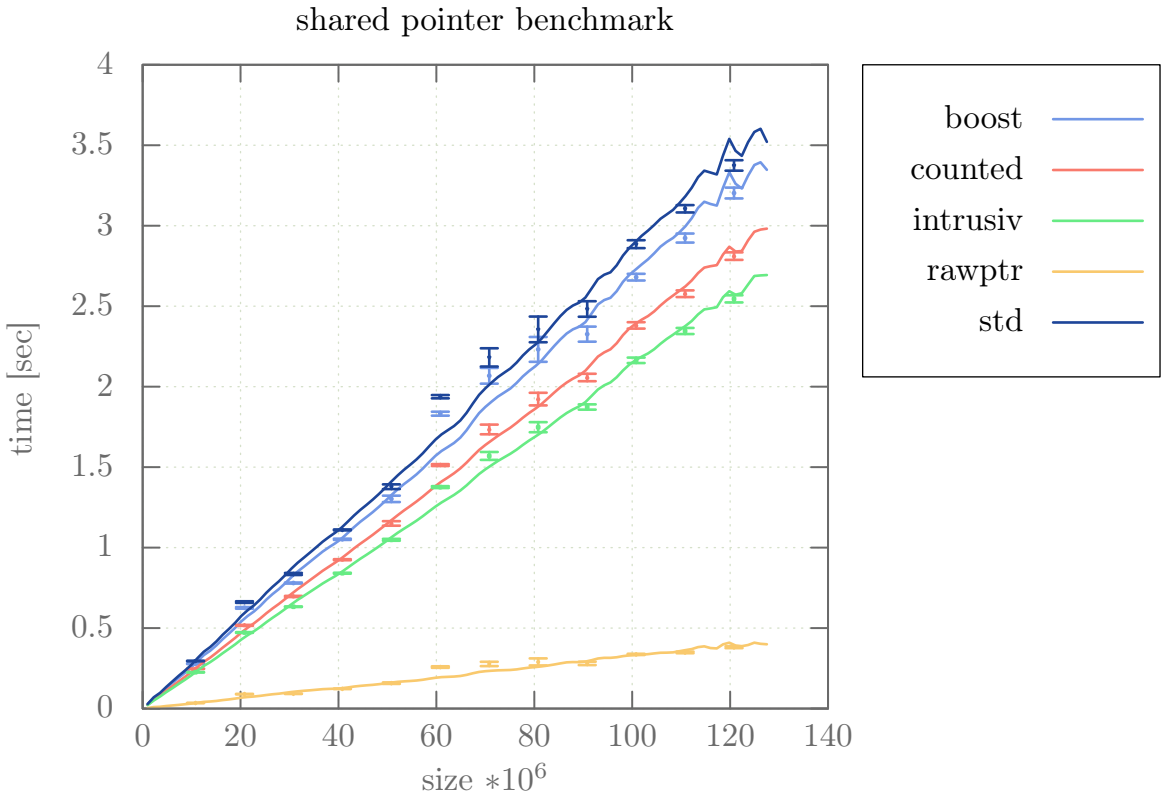
- Shared Pointer
- Referenzzählung → sync Counter

Smart Pointer- Benchmark



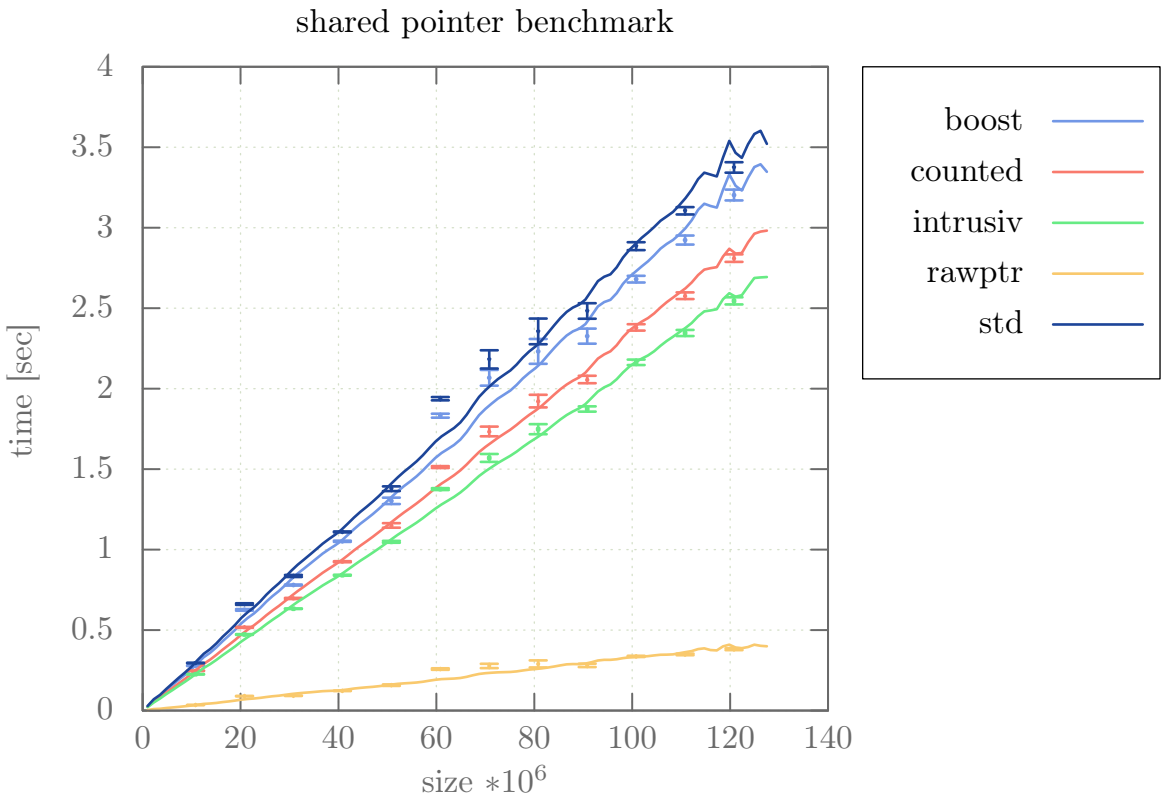
- Szenario: zweifacher Zugriff
- Varianten (RawPointer, intrusive)
- builddb: $\approx$ 37% Gewinn, camelod: $>$ 30% (Scans)

Smart Pointer- Benchmark



- Szenario: zweifacher Zugriff
- Varianten (RawPointer, intrusive)
- builddb: $\approx$ 37% Gewinn, camelod: $>$ 30% (Scans)

Smart Pointer- Benchmark



- Szenario: zweifacher Zugriff
- Varianten (RawPointer, intrusive)
- builddb: $\approx$ 37% Gewinn, camelod: $>$ 30% (Scans)

RLE Kompression - Verfahren

- Kompression: Chunk-Vektoren
- $S = [aaaaabbbbbbbccccccc]$, mit $a \neq b \neq c$
- $M = [abc]$
- $P = [0, 5, 13]$
- $P \rightarrow$ *uShort*-Sequenz, sortiert
- $size(S) = 18 \cdot 8\ Byte = 144\ Byte$
- $size(M + P) = 3 \cdot 8\ Byte + 3 \cdot 2\ Byte = 30\ Byte$
- Blockgröße datenabhängig $\rightarrow$ Messung
- $\alpha = |M|/|S| \approx 0.16$

RLE Kompression - Verfahren

- Kompression: Chunk-Vektoren
- $S = [aaaaabbbbbbbbccccccc]$, mit $a \neq b \neq c$
- $M = [abc]$
- $P = [0, 5, 13]$
- $P \rightarrow$ *uShort*-Sequenz, sortiert
- $size(S) = 18 \cdot 8\ Byte = 144\ Byte$
- $size(M + P) = 3 \cdot 8\ Byte + 3 \cdot 2\ Byte = 30\ Byte$
- Blockgröße datenabhängig $\rightarrow$ Messung
- $\alpha = |M|/|S| \approx 0.16$

RLE Kompression - Verfahren

- Kompression: Chunk-Vektoren
- $S = [aaaaabbbbbbbbccccccc]$, mit $a \neq b \neq c$
- $M = [abc]$
- $P = [0, 5, 13]$
- $P \rightarrow$ *uShort*-Sequenz, sortiert
- $size(S) = 18 \cdot 8\ Byte = 144\ Byte$
- $size(M + P) = 3 \cdot 8\ Byte + 3 \cdot 2\ Byte = 30\ Byte$
- Blockgröße datenabhängig $\rightarrow$ Messung
- $\alpha = |M|/|S| \approx 0.16$

RLE Kompression - Verfahren

- Kompression: Chunk-Vektoren
- $S = [aaaaabbbbbbbbccccccc]$, mit $a \neq b \neq c$
- $M = [abc]$
- $P = [0, 5, 13]$
- $P \rightarrow$ *uShort*-Sequenz, sortiert
- $size(S) = 18 \cdot 8\ Byte = 144\ Byte$
- $size(M + P) = 3 \cdot 8\ Byte + 3 \cdot 2\ Byte = 30\ Byte$
- Blockgröße datenabhängig $\rightarrow$ Messung
- $\alpha = |M|/|S| \approx 0.16$

RLE Kompression - Verfahren

- Kompression: Chunk-Vektoren
- $S = [aaaaabbbbbbbbccccccc]$, mit $a \neq b \neq c$
- $M = [abc]$
- $P = [0, 5, 13]$
- $P \rightarrow$ *uShort*-Sequenz, sortiert
- $size(S) = 18 \cdot 8\ Byte = 144\ Byte$
- $size(M + P) = 3 \cdot 8\ Byte + 3 \cdot 2\ Byte = 30\ Byte$
- Blockgröße datenabhängig $\rightarrow$ Messung
- $\alpha = |M|/|S| \approx 0.16$

RLE Kompression - Verfahren

- Kompression: Chunk-Vektoren
- $S = [aaaaabbbbbbbbccccccc]$, mit $a \neq b \neq c$
- $M = [abc]$
- $P = [0, 5, 13]$
- $P \rightarrow$ *uShort*-Sequenz, sortiert
- $size(S) = 18 \cdot 8\ Byte = 144\ Byte$
- $size(M + P) = 3 \cdot 8\ Byte + 3 \cdot 2\ Byte = 30\ Byte$
- Blockgröße datenabhängig $\rightarrow$ Messung
- $\alpha = |M|/|S| \approx 0.16$

RLE Kompression - Verfahren

- Kompression: Chunk-Vektoren
- $S = [aaaaabbbbbbbbccccccc]$, mit $a \neq b \neq c$
- $M = [abc]$
- $P = [0, 5, 13]$
- $P \rightarrow$ *uShort*-Sequenz, sortiert
- $size(S) = 18 \cdot 8\ Byte = 144\ Byte$
- $size(M + P) = 3 \cdot 8\ Byte + 3 \cdot 2\ Byte = 30\ Byte$
- Blockgröße datenabhängig $\rightarrow$ Messung
- $\alpha = |M|/|S| \approx 0.16$

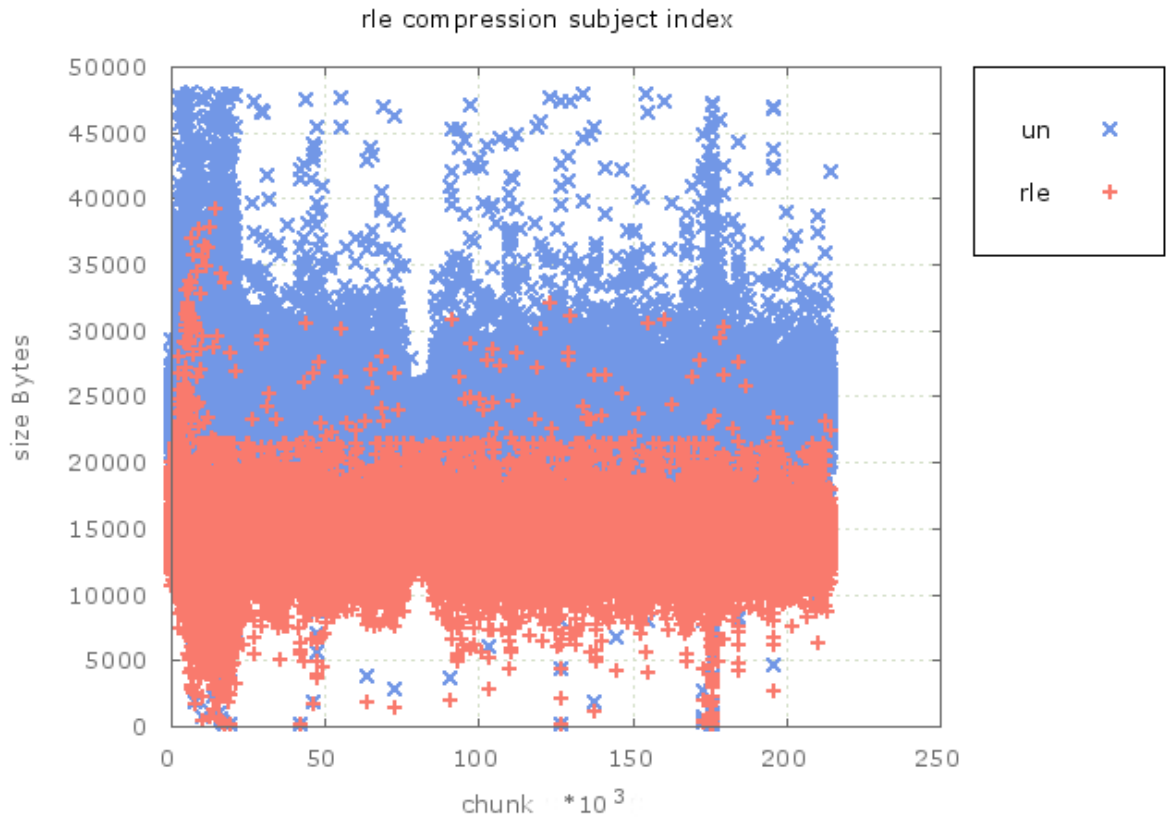
RLE Kompression - Verfahren

- Kompression: Chunk-Vektoren
- $S = [aaaaabbbbbbbbccccccc]$, mit $a \neq b \neq c$
- $M = [abc]$
- $P = [0, 5, 13]$
- $P \rightarrow$ *uShort*-Sequenz, sortiert
- $size(S) = 18 \cdot 8\ Byte = 144\ Byte$
- $size(M + P) = 3 \cdot 8\ Byte + 3 \cdot 2\ Byte = 30\ Byte$
- Blockgröße datenabhängig $\rightarrow$ Messung
- $\alpha = |M|/|S| \approx 0.16$

RLE Kompression - Verfahren

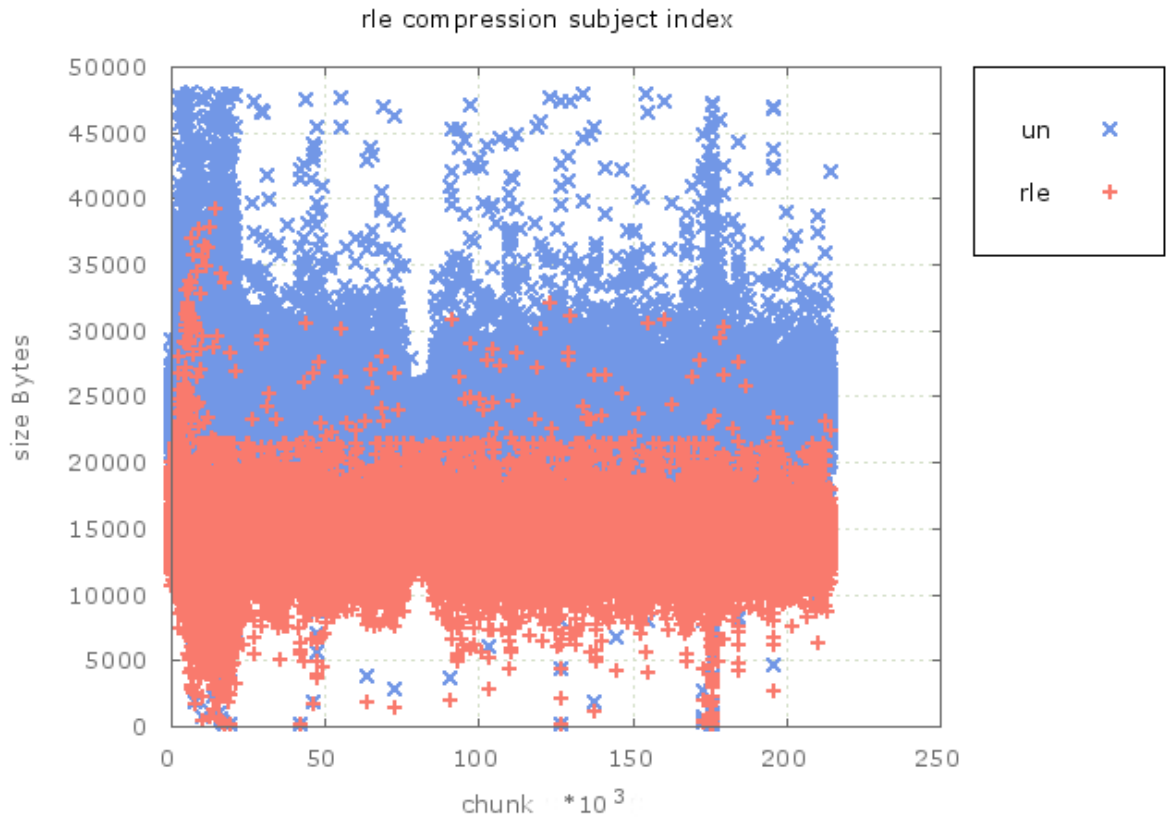
- Kompression: Chunk-Vektoren
- $S = [aaaaabbbbbbbbccccccc]$, mit $a \neq b \neq c$
- $M = [abc]$
- $P = [0, 5, 13]$
- $P \rightarrow$ *uShort*-Sequenz, sortiert
- $size(S) = 18 \cdot 8\ Byte = 144\ Byte$
- $size(M + P) = 3 \cdot 8\ Byte + 3 \cdot 2\ Byte = 30\ Byte$
- Blockgröße datenabhängig $\rightarrow$ Messung
- $\alpha = |M|/|S| \approx 0.16$

RLE Kompression - Messung



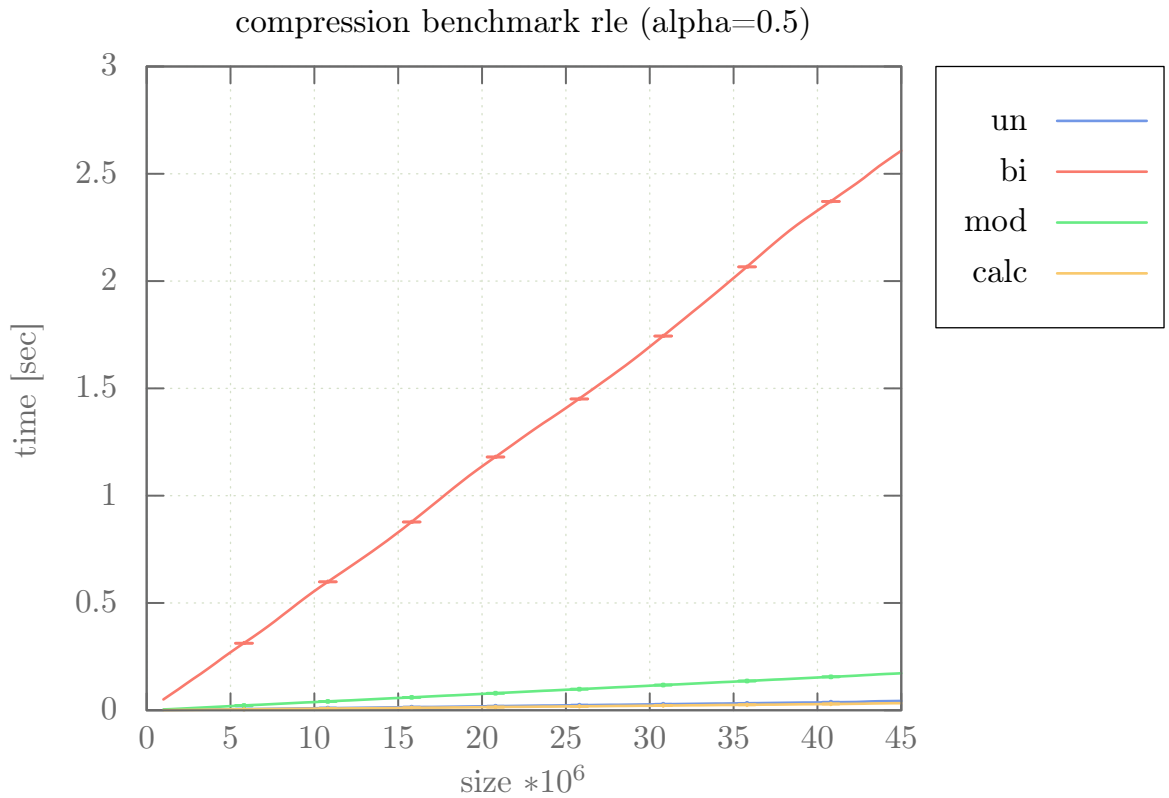
- Analyse der Chunks der Indizes
- α_{avg} s:0.47 p:0.45 o:0.52
- *compRatio* s:$\approx$ 0.58 p:$\approx$ 0.56 o:$\approx$ 0.64
- $\rightarrow$ 14.4 *GB* $\rightarrow$ 8.6 *GB*

RLE Kompression - Messung



- Analyse der Chunks der Indizes
- α_{avg} s:0.47 p:0.45 o:0.52
- *compRatio* s:$\approx$ 0.58 p:$\approx$ 0.56 o:$\approx$ 0.64
- $\rightarrow$ 14.4 *GB* $\rightarrow$ 8.6 *GB*

RLE Kompression - Zugriffe/Benchmark



- Testszenario = Summation von Elementen
- $\alpha = 0.5 =$ Auslastung
- verschiedene Zugriffsmethoden (bi, mod, calc, un)

Fazit

- Evaluierung $\surd$
- Verbesserungen (Laufzeit/Speicher) $\surd$

Ausblick

- Integration (RLE, FOR,...)
- Memory Kontext
- Chunk/Spalten-Typen
- MMF Konzepte (Speicherung)
- TBB vs NUMA

Fazit

- Evaluierung $\surd$
- Verbesserungen (Laufzeit/Speicher) $\surd$

Ausblick

- Integration (RLE, FOR,...)
- Memory Kontext
- Chunk/Spalten-Typen
- MMF Konzepte (Speicherung)
- TBB vs NUMA

Fazit

- Evaluierung $\surd$
- Verbesserungen (Laufzeit/Speicher) $\surd$

Ausblick

- Integration (RLE, FOR,...)
- Memory Kontext
- Chunk/Spalten-Typen
- MMF Konzepte (Speicherung)
- TBB vs NUMA

Fazit

- Evaluierung $\surd$
- Verbesserungen (Laufzeit/Speicher) $\surd$

Ausblick

- Integration (RLE, FOR,...)
- Memory Kontext
- Chunk/Spalten-Typen
- MMF Konzepte (Speicherung)
- TBB vs NUMA

Fazit

- Evaluierung $\surd$
- Verbesserungen (Laufzeit/Speicher) $\surd$

Ausblick

- Integration (RLE, FOR,...)
- Memory Kontext
- Chunk/Spalten-Typen
- MMF Konzepte (Speicherung)
- TBB vs NUMA

Fazit

- Evaluierung $\surd$
- Verbesserungen (Laufzeit/Speicher) $\surd$

Ausblick

- Integration (RLE, FOR,...)
- Memory Kontext
- Chunk/Spalten-Typen
- MMF Konzepte (Speicherung)
- TBB vs NUMA

Fazit

- Evaluierung $\surd$
- Verbesserungen (Laufzeit/Speicher) $\surd$

Ausblick

- Integration (RLE, FOR,...)
- Memory Kontext
- Chunk/Spalten-Typen
- MMF Konzepte (Speicherung)
- TBB vs NUMA

Fragen?

Vielen Dank für Ihre Aufmerksamkeit.