Kapitel 1

Sammanfattning

I denna laboration undersöktes en hashtabells prestanda då tre olika hashfunktioner användes. De tre hashfunktionerna evaluerades med ett testprogram som mätte antal kollisioner, tid för uppslagning vid olika storlekar på hashtabellen, och tiden för uppslagning vid olika antal insatta element i tabellen. Testerna utfördes för fem olika storlekar på hashtabellen. Testerna visade att hashfunktionen *djb2* både var snabbast och gav minst antal kollisioner, samt att en hashtabellstorlek på 2^{21} räckte för att erhålla en effektiv krockhantering.

Utöver dessa tester undersöktes även en naiv hashfunktion som använde sig av ascii-värden för att beräkna ett numeriskt hashvärde. I fallet med den naiva hashfunktionen evaluerades spridningen av kollisioner för tre olika skiftkonstanter. Resultaten visade att fördelningen av kollisioner över hashtabellen var bäst då skiftkonstanten sattes till 5.

Kapitel 2

Formalia

Fabian Revilla
1995-03-08
DD1321 Tillämpad programmering och datalogi - KTH Juni 2019

Kapitel 3

Uppgiftsbeskrivning

Denna labben bygger vidare på en hashtabellsimplementation i programspråket C från en tidigare labb i kursen Tillämpad programmering och datalogi på KTH. Hashtabellens prestanda evalueras genom att olika parametrar såsom val av hashfunktion och storlek på hashtabellen varieras. I uppgiften används filen *tab_tracks.txt*, som innehåller data över en miljon låtar, där varje element består av låtnamn och artistnamn.

3.0.1 Testprogrammets struktur

Det givna testprogrammet test.c läser in filen och anropar funktionen put en gång per låt, med artisten som nyckel och låtnamnet som värde. Koden i test.c, som innehåller en mainfunktion, gör följande.

- Låtlistan läses in från filen *tab_tracks.txt*. Varje låt sätts in i hashtabellen med put. När ett jämnt tusental låtar har satts in körs hundra anrop till get, och tidsåtgången för det sparas i filen resultat/*-uppslagning.dat (där * ersätts av hashalgoritmen).
- Efter att alla låtar har lagts in görs en provuppslagning av nyckeln (artisten) Pet Shop Boys, så att programmet kan kontrollera att det korrekta värdet Miserablism (2001 Digital Remaster) returnerades.
- Programmet går igenom hela hashtabellen och räknar antalet kollisioner. Det är summan av längden av alla krocklistor, fast minus ett på varje lista eftersom en lista med ett element inte utgör en kollision. Varje gång programmet stöter på en lista med fler än ett element – varje gång det har skett en kollision – sparas indexet för den listan tillsammans med listans längd ned i filen resultat/*-kollisioner.dat.
- Efter kollisionsräkningen kommer programmet genomföra upprepade antal anrop till get för en specifik nyckel. Först sker 10 000 anrop, sedan 20 000 osv. Tidsåtgången sparas ned i filen resultat/*-uppslagningar.dat.

Kapitel 4

Metod

4.0.1 Jämförelse av hashfunktioner

Initialt kompilerades test.c med filer från de tidigare labbarna på följande sätt:

```
$ gcc lista.c hashfunc.c test.c hashfunktioner/tilpro-modulo.c
$ gcc lista.c hashfunc.c test.c hashfunktioner/[...].c
```

Figur 4.1

Där hakparenteserna på den nedre raden byttes ut mot en av de tre hashfunktionerna *tilpro-bitand*, *tilpro-modulo* och *djb2*. Operationen resulterade i en exekverbar fil som vid körning genererade tre .dat filer, innehållandes data över hashunktionens prestanda. Proceduren upprepades för varje hashfunktion, varefter totalt 9 .dat-filer erhölls.

För att jämföra de olika hashfunktionernas prestanda kördes skriptet ”kör-experimenten.sh”via terminalen. Skriptet genererade fem stycken PNG-bilder med kombinerat resultat över de tre hashfunktionernas prestanda. Testet upprepades för fem olika storlekar på hastabellen. De hashstorlekar som användes var alla jämna tvåpotenser och redovisas i bilden nedan, där hashtabellens storlek representeras av konstanten HASHVEKSIZE.

```
//Konstanter
const int HASHVEKSIZE = 1048576; // 2 upphöjt till 20 ungefär 1 miljon
//const int HASHVEKSIZE = 2097152;// 2 upphöjt till 21
//const int HASHVEKSIZE = 4194304;// 2 upphöjt till 22
//const int HASHVEKSIZE = 8388608;// 2 upphöjt till 23
//const int HASHVEKSIZE = 16777216;// 2 upphöjt till 24
```

Figur 4.2

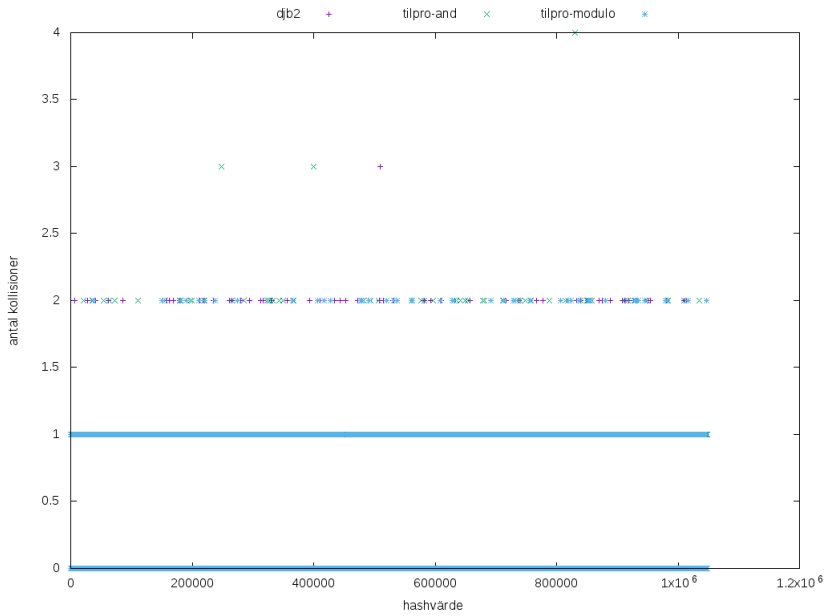
4.0.2 Kollisionsspridning med en naiv hashfunktion

Vid tester av den naiva hashfunktionen satte HASHVEKSIZE till 2^{21}, eftersom den storleken teoretiskt sett ska generera en tabell med effektiv krockhantering. För att utföra testerna

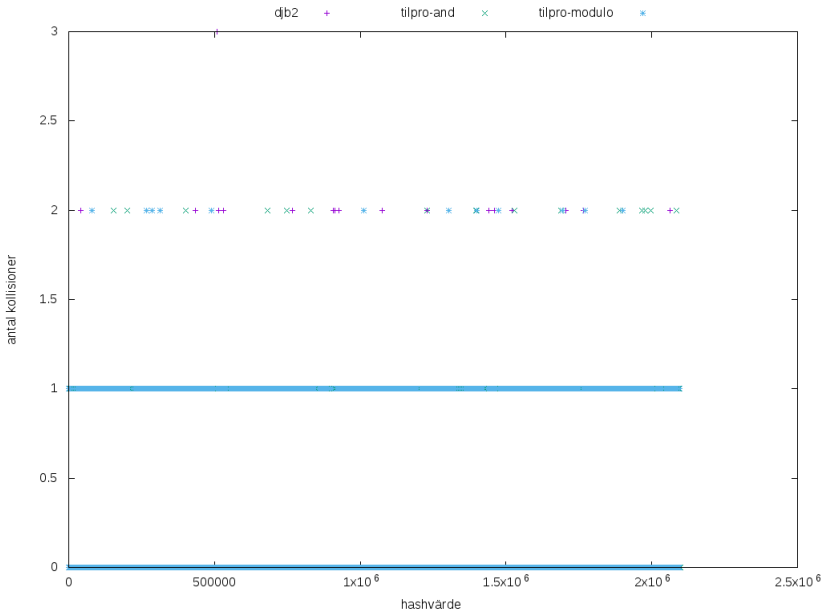
Kapitel 5

Resultat

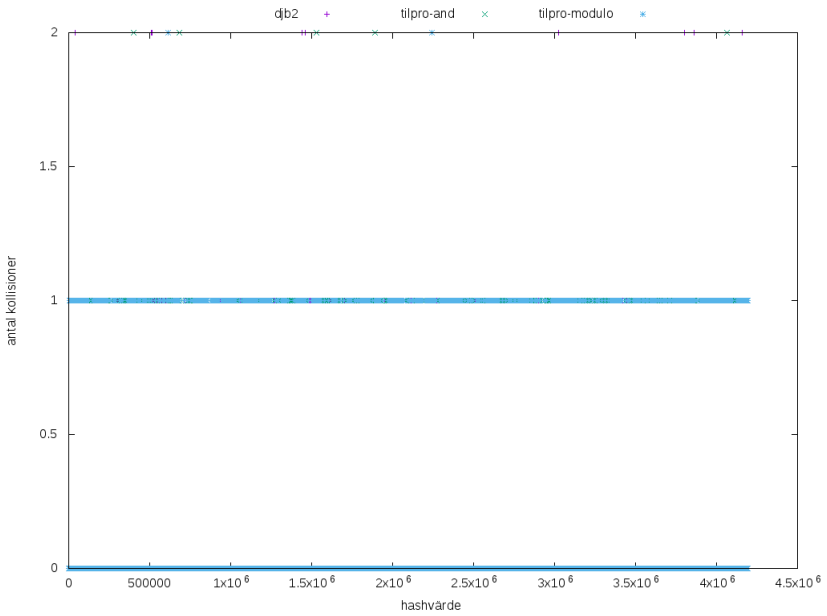
5.0.1 Antal kollisioner plottat över hashvärde



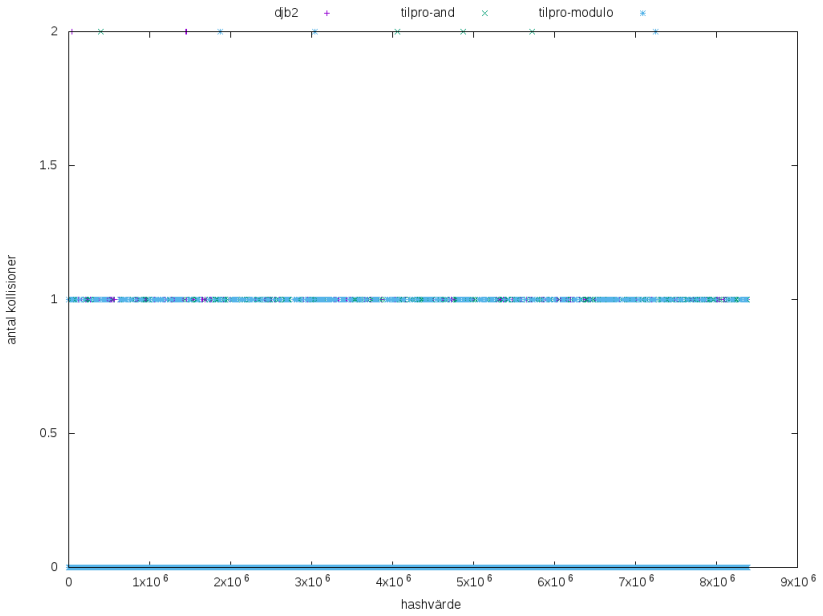
Figur 5.1: HASHVEKSIZE = 2^{20}



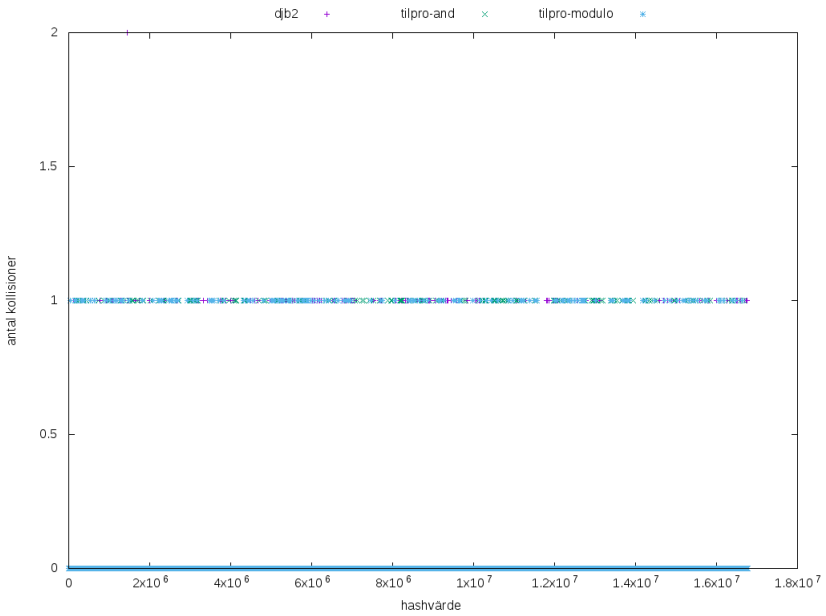
Figur 5.2: HASHVEKSIZE = 2^{21}



Figur 5.3: HASHVEKSIZE = 2^{22}

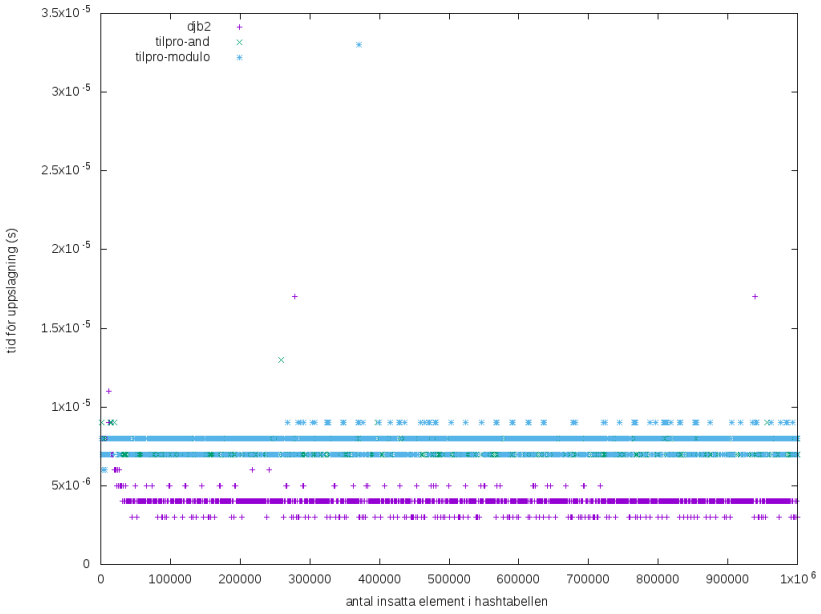


Figur 5.4: HASHVEKSIZE = 2^{23}

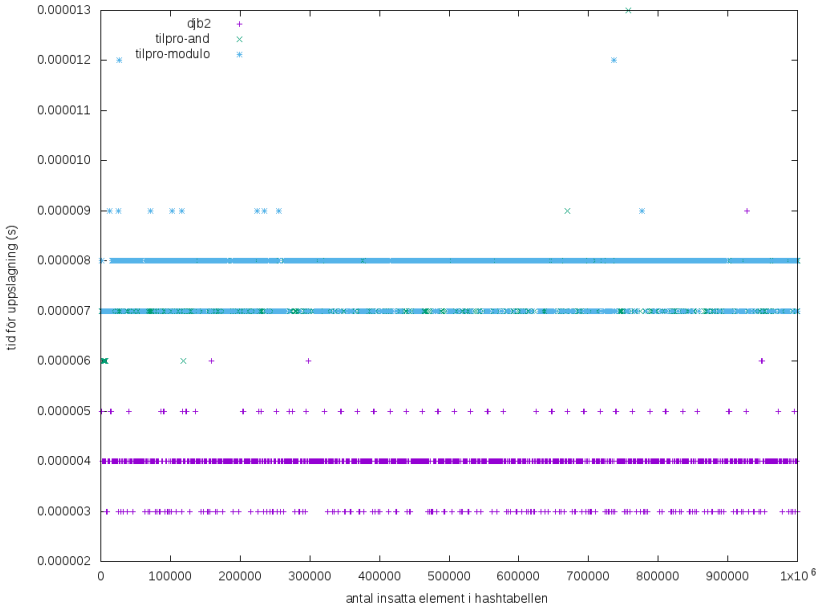


Figur 5.5: HASHVEKSIZE = 2^{24}

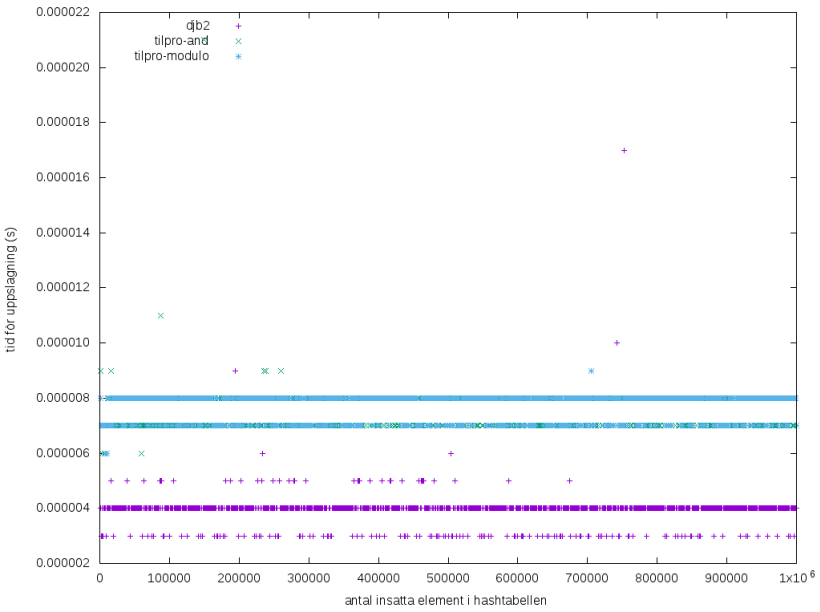
5.0.2 Tid för uppslagning plottat mot antal insatta element



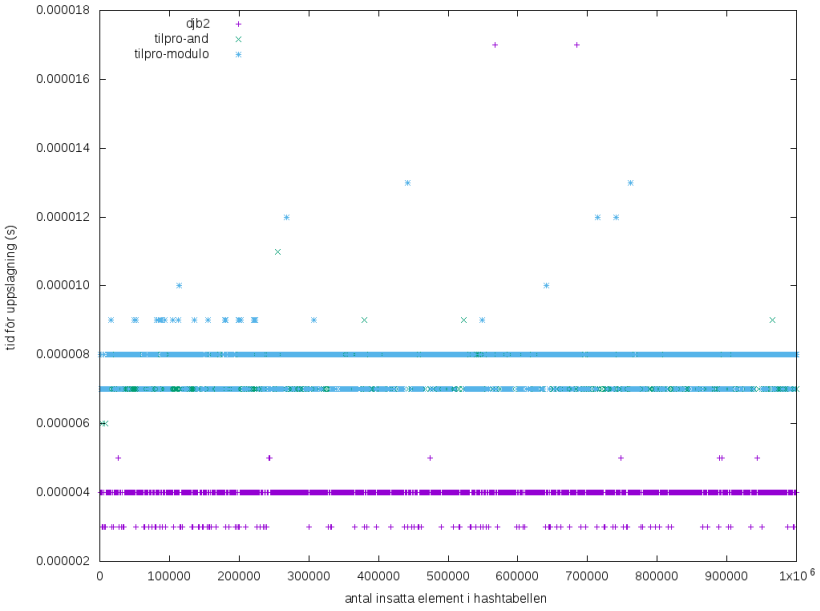
Figur 5.6: HASHVEKSIZE = 2^{20}



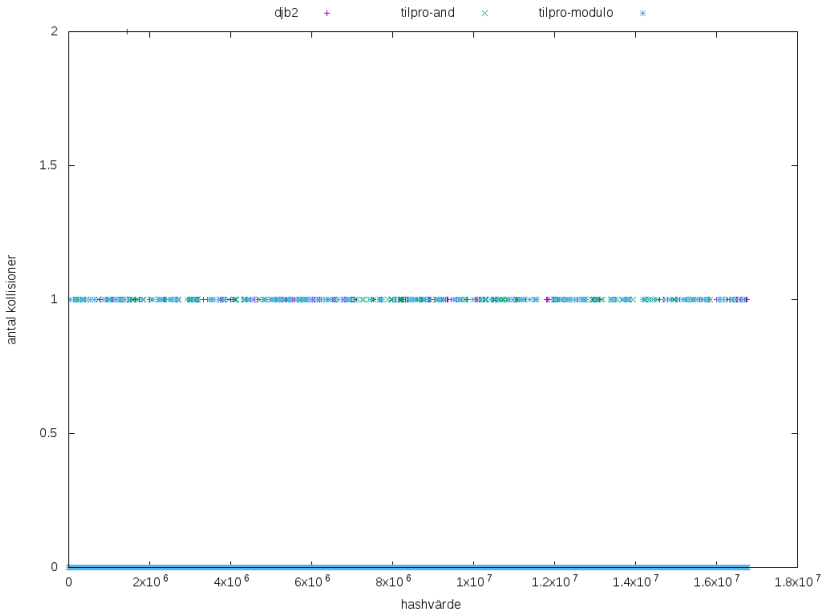
Figur 5.7: HASHVEKSIZE = 2^{21}



Figur 5.8: HASHVEKSIZE = 2^{22}

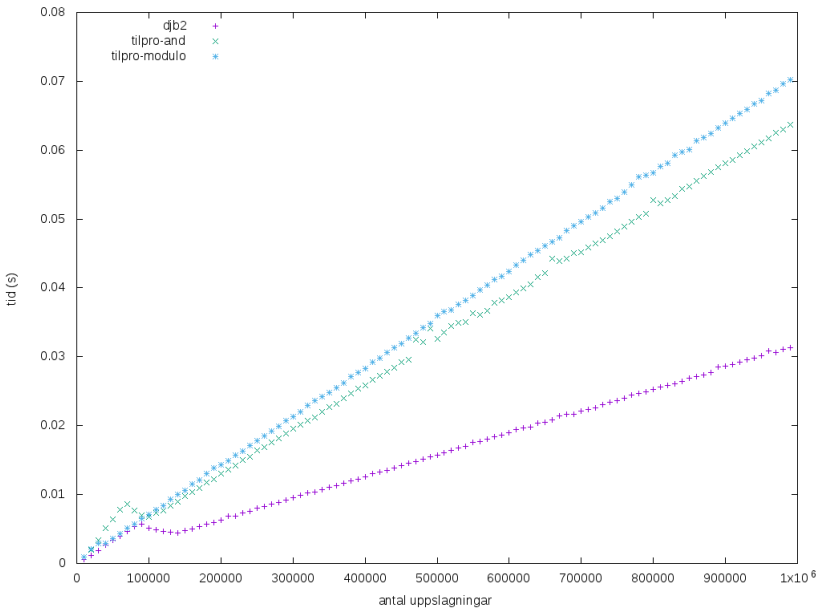


Figur 5.9: HASHVEKSIZE = 2^{23}

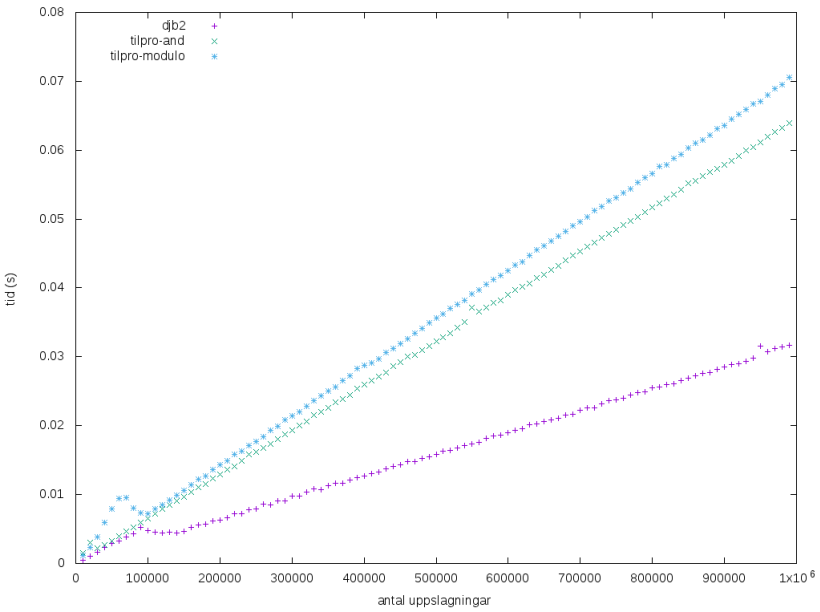


Figur 5.10: HASHVEKSIZE = 2^{24}

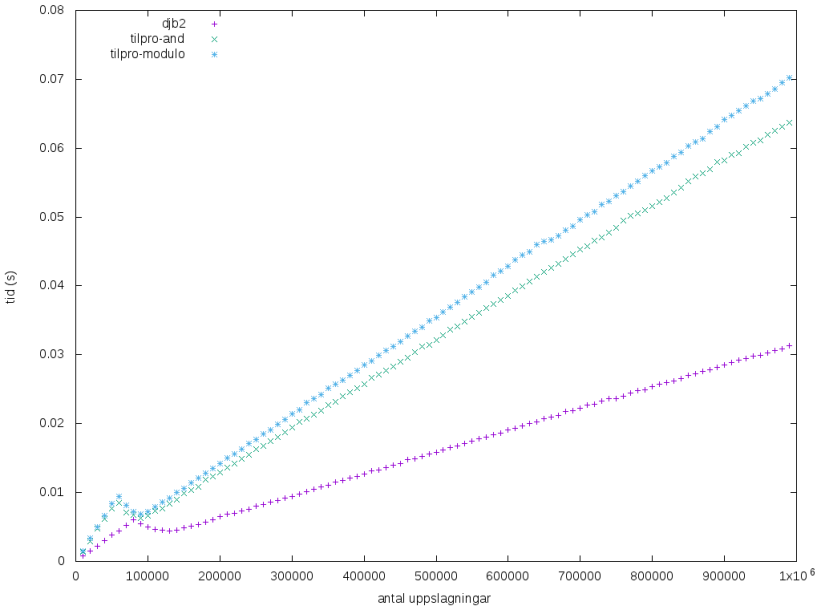
5.0.3 Tidsåtgång plottat mot antal uppslagningar



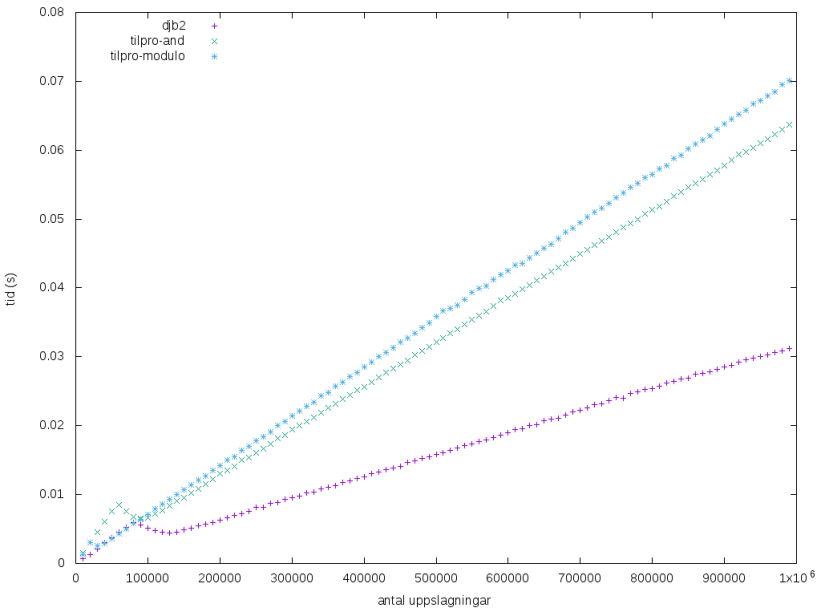
Figur 5.11: HASHVEKSIZE = 2^{20}



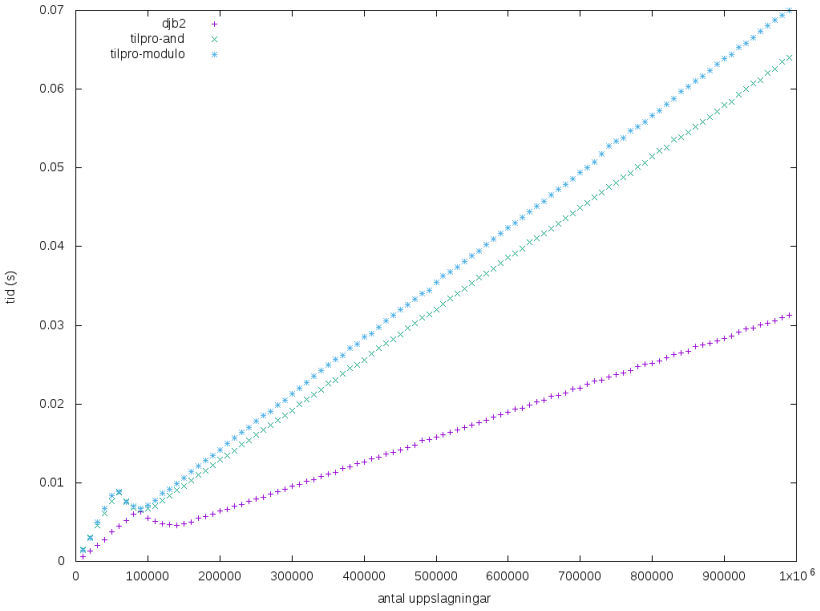
Figur 5.12: HASHVEKSIZE = 2^{21}



Figur 5.13: HASHVEKSIZE = 2^{22}

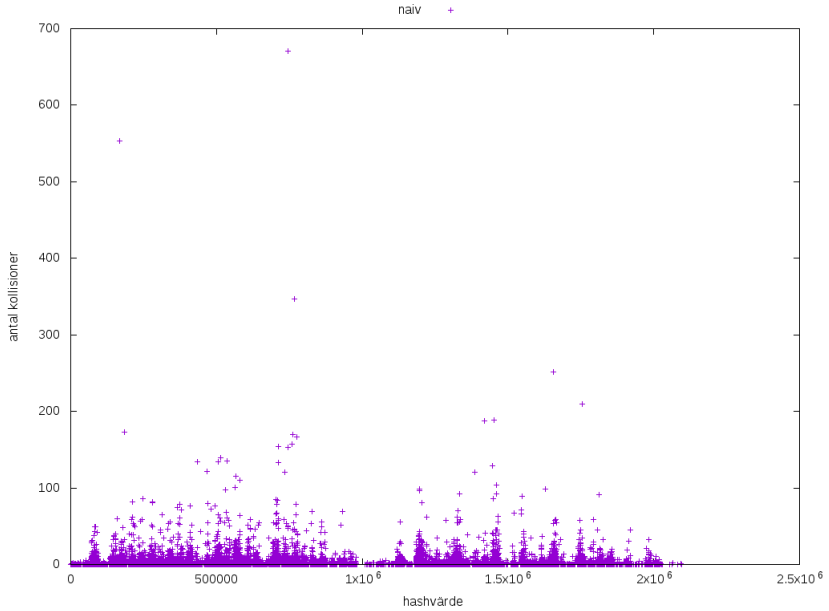


Figur 5.14: HASHVEKSIZE = 2^{23}

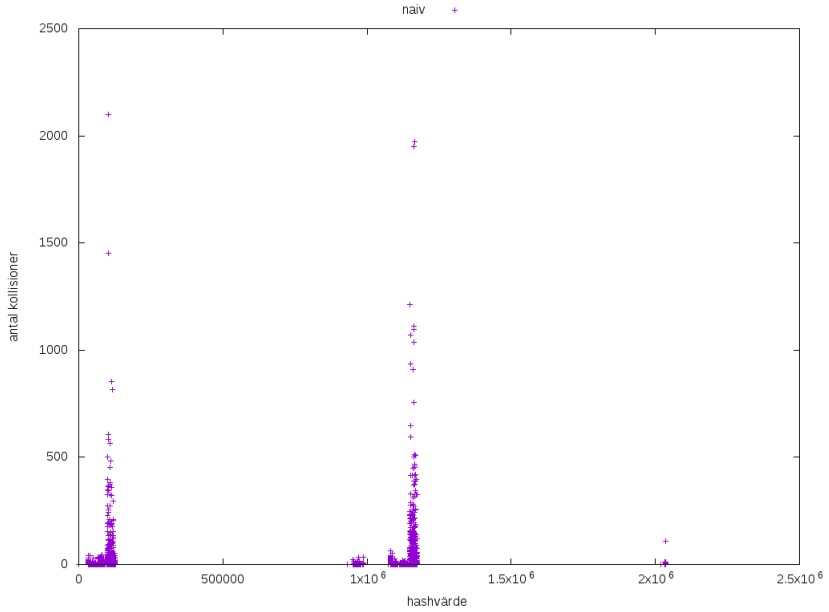


Figur 5.15: HASHVEKSIZE = 2^{24}

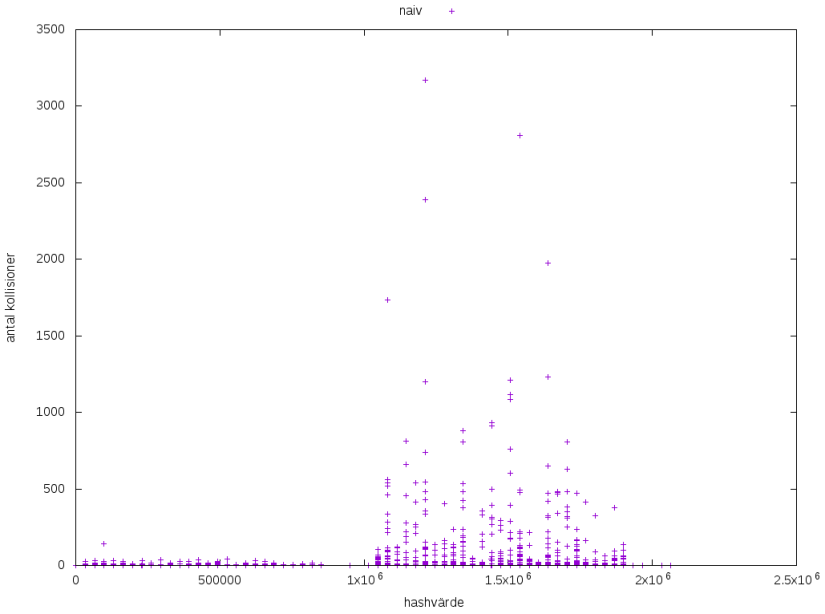
5.0.4 Antal kollisioner med en naiv hashfunktion



Figur 5.16: Skiftkonstant = 5



Figur 5.17: Skiftkonstant = 10



Figur 5.18: Skiftkonstant = 15

Kapitel 6

Analys

I detta kapitel besvaras de givna frågeställningarna.

- Ger någon av hashfunktionerna bättre eller sämre prestanda än någon annan? I så fall, är det genomgående så?

  När det gäller snabbhet går det att konstatera att *djb2* genomgående presterar bättre än de andra hashfunktionerna, oberoende av storleken på hashtabellen. Detta exemplifieras av bl.a. Figur 5.6 och Figur 5.11.På samtliga plottar av tidsåtgång mot antal uppslagningar, handlar det om en skillnad på ungefär tre hundradelar av en sekund. Mellan *tilpro-modulo* och tilpro-and är det smärre skillnad i tidseffektivitet, vilket kan ses på Figur 5.6 *djb2* gav även väsentligt färre krockar än *tilpro-modulo* och ungefär lika många som tilpro-and.

- Hur påverkas uppslagninstiden av antalet element i hashtabellen (uppslagning.png), kontra hashtabellens storlek? Vad hade du teoretiskt kunnat förvänta dig?

  Som konstaterat tidigare är *djb2* den snabbaste hashfunktionen enligt testerna. En föklaring till detta är att *djb2* innehåller färre bitoperationer än de handra hashfunktionerna och därför opererar den snabbare.

  Uppslagstiden verkar påverkas i väldigt liten utsträckning av antalet insatta element. Enligt teorin är uppslag ur en hashtabell i genomsnitt av ordningen O(1). Endast i sällsynta fall, då det sökta värdet inte finns på tänkta platsen och tabellen måste fortsättas sökas igenom, är den långsammare. Eftersom alla låtar får plats i tabellen och nycklarna sprids effektivt över hashtabellen, sker inte så många krockar. Att plocka ut element ur tabellen kommer därför kosta lika mycket tid oavsett hur många element som finns lagrade i den.

- Vad vet du om hur stor en hashtabell bör vara och hur väl tycks det stämma i praktiken?

  Enligt teorin bör en hashtabell ha 50% luft för att erhålla en effektiv krockhantering. I detta fall skulle det innebära en vektorstorlek på minst två miljoner(då antalet låtar som ska hashas är en miljon). När hashtabellens storlek

ändras från 220 (Figur 5.1 Figur 5.2)till 2^{22}(ca 2 mil), går det att tyda att antalet krockar minskar. Dock är denna skillnaden inte mycket större än mellan hashtabellerna med storlek 2^{21}, respektive 2^{22}. Detta understruker att denna regel är mer av en teoretisk riktlinje än ett exakt värde.

- Hur påverkas uppslagningen av hashtabellens storlek? Varför?

  Tiden för uppslagning verkar ligga ungefär på samma nivå för alla hashfunktioner, oberoende av hashtabellens storlek. tilpro-modulu tenderar att alternera mellan 8 och 7 mikrosekunder, medan *djb2* mestadels ligger vid 4 mikrosekunder och tilpro-and konsekvent ligger runt 7 mikrosekunder. När hashtabellen är mindre än 2 miljoner, går det dock att se spridningen är större, vilket också medför att den första plotten är mer utdragen i höjdled. Detta kan bero på att antalet krockar, som konstaterat innan, är större då hashtabellens inte har 50% luft. Detta meför att det i genomsnitt tar längre tid att slå upp värden ur hashtabellen.

- Vad är det för skillnad på *tilpro-bitand* och *tilpro-modulo*, och vad får skillnaden för effekt i praktiken (i körningarna)? Varför?

  Det enda som skiljer funktionerna åt rent kodmässigt en av de sista raderna där &-tecknet i and-funktionen har bytits ut mot % i modulo-funktionen. I den förstnämnda görs alltså en logisk AND-operation med variabeln hash och HASHVEKSIZE, istället för en modulo-operation.

```
hash = hash % (HASHVEKSIZE - 1);
```

```
hash = hash & (HASHVEKSIZE - 1);
```

Figur 6.1: Skillnaden mellan tilpro-and och *tilpro-modulo*

  En tydlig trend är tilpro-and presterar bättre än *tilpro-modulo* på de tidsrelaterade testerna. En föklaring till detta kan vara att en AND-operation mellan två bitvärden helt enkelt är en enklare räkneoperation, medan modulo är en inbyggd funktion som är mer komplext uppbyggd då den innehåller fler steg.

- Vad är det för skillnad i spridningen av kollisioner? Hur kommer det sig?

  Förutom att det totala antalet krockar är mycket större hos den naiva hashfunktionen går även det att tyda att det finns periodiska trender i spridningen av kollisioner. Detta syns tydligt i figur Figur 5.17, då skiftkonstanten sattes till 10.

- I utskriften från test.c får man kollisionslistan för Pet Shop Boys, dvs. alla nycklar som har hashats in med samma hashvärde som Pet Shop Boys. Vad kan du dra för slutsatser utifrån den listan och källkoden för naiv.c?

Alla nycklar i kollisionslista verkar ha olika längd. Den naiva hashfunktionen beräknar hashvärdet genom att summera ascii-värdet för varje separat karaktär multiplicerat med en potens av 32. Detta kommer leda till att många olika nycklar(artistnamn) kan summeras till samma hashvärde. Detta leder oundvikligen till en hel del krockar.

- Vad händer med fördelningen av kollisioner om du ökar eller minskar på skiftkonstanten i naiv.c? Vad gör skiftkonstanten, egentligen? Tips: Tänk på asciitabellen!

  Skiftkonstantet förflyttar en bit(ett tal) till vänster, vilket i praktiken innebär att en multiplikation sker. Om en bit skiftas fem steg åt vänster multipliceras det givna talet med $2^5 = 32$ i tiotalssystemet. När skiftkonstanten i detta fall ökades visade det sig att kollisionerna klumpade ihop sig och spridningen blev mer ojämnt fördelad över hashtabellen, vilken kan ses på Figur 5.17

- Vad är en önskvärd fördelning av kollisioner, och kan du hitta en skiftkonstant som gör att du närmar dig den fördelningen?

  En önskvärd fördelning är då kollisionerna fördelas mer eller mindre jämnt över hela hashtabellen. Som slutsats av de tre tester som utfördes går det att konstatera att när skiftkonstanten sattes till 5 så var fördelningen mycket mer jämn än i de andra fallen. För att hitta en mer optimal skiftkonstant skulle fler tester behövas utföras.