سوالات بخش Index Compression

سوال ۱

چرا در فشرده‌سازی postings، قبل از هر چیز docIDها را به شکل gap (شکاف) تبدیل می‌کنیم؟
در پاسخ خودت حتما بگو:

- چه خاصیتی در لیست postings باعث می‌شود تعریف gap معنی‌دار باشد.
- چرا برای واژه‌های پُرتکرار، مقدار این gapها معمولا کوچک است.
- کوچک بودن gapها چطور به نفع روش‌های کدگذاری مثل Variable-Byte و Gamma می‌شود.

سوال ۲

کدگذاری Variable-Byte با اندازه پکت $b=8$ را توضیح بده.
در پاسخ:

1. نقش «بیت ادامه» (continuation bit) چیست و چه مقادیری می‌گیرد؟
2. ۷ بیت بعدی در هر پکت چه کاری انجام می‌دهند؟
3. از چه جهت می‌گوییم این روش از UTF-8 الهام گرفته شده است، و چرا استفاده مستقیم از UTF-8 برای فشرده‌سازی postings نوعی over-engineering محسوب می‌شود؟

سوال ۳

لیست مرتب docID زیر را برای یک واژه در نظر بگیر:

[56, 58, 63, 64, 67]

1. این لیست را به لیست gap تبدیل کن.
2. الگوریتم کلی کدگذاری یک عدد مثبت N با Variable-Byte و $b=8$ را، مرحله به مرحله (بدون محاسبه باینری دقیق)، توضیح بده.
3. توضیح بده دیکودر چگونه می‌تواند بدون داشتن هیچ اشاره‌گر اضافی، gapها را از روی دنباله بایت‌ها بازیابی کند.

سوال ۴

دو پیاده‌سازی متفاوت از Variable-Byte را در نظر بگیر:

- طرح A: پکت‌ها با اندازه $b=4$ بیت (۱ بیت ادامه + ۳ بیت داده).
- طرح B: پکت‌ها با اندازه $b=8$ بیت (۱ بیت ادامه + ۷ بیت داده).

به پرسش‌های زیر پاسخ بده:

1. برای چه نوع توزیع gap (اکثرا کوچک یا اکثرا بزرگ بودن gapها) طرح A از نظر تعداد بیت، فشرده‌تر از طرح B خواهد بود؟ چرا؟
2. در چه حالتی انتظار داری طرح B از نظر سرعت دیکود (CPU کمتر) بهتر از طرح A عمل کند؟
3. این تفاوت‌ها را به‌طور کوتاه به مبحث کلی «تجارت بین I/O و CPU» در فشرده‌سازی شاخص ربط بده.

پاسخ ها:

پاسخ سوال ۱

- لیست postings برای هر واژه به‌صورت صعودی مرتب از نظر docID است، بنابراین اختلاف بین دو docID متوالی همیشه غیرمنفی و معنی‌دار است (gap).
- در واژه‌های پُرتکرار، همان واژه در سندهای زیادی ظاهر می‌شود و این سندها از نظر شناسه به هم «نزدیک» هستند، پس اختلاف بین docIDهای متوالی (gapها) اغلب عددهای کوچک مثل ۱، ۲، ۳ و... می‌شود.
- وقتی gapها کوچک‌اند، روش‌های طول‌متغیر مثل Variable-Byte و Gamma می‌توانند این اعداد کوچک را با بیت‌های خیلی کم کد کنند، در نتیجه اندازه کلی postings به‌طور چشمگیری کاهش می‌یابد.

پاسخ سوال ۲

1. در Variable-Byte با b=8b=8، بیت اول هر بایت «بیت ادامه» است: اگر ۰ باشد یعنی این بایت هنوز وسط نمایش همان عدد است و بایت‌های دیگری نیز برای این عدد خواهند آمد؛ اگر ۱ باشد، یعنی این بایت آخرین بایت نمایش آن عدد است.
2. ۷ بیت بعدی، بخش‌هایی از نمایش باینری عدد را (payload) حمل می‌کنند؛ وقتی چند بایت پشت سر هم برای یک عدد داریم، ۷ بیت‌های آن‌ها به ترتیب کنار هم قرار می‌گیرند تا باینری کامل عدد به‌دست آید.
3. این طرح از UTF-8 الهام گرفته، چون در UTF-8 هم طول کد هر کاراکتر متغیر است و بیت‌های ابتدایی هر بایت نقش هدر را دارند و می‌گویند چند بایت مربوط به همان کاراکتر است؛ اما برای postings نیاز نداریم همه الگوهای پیچیده UTF-8 (مثل 1110xxxx ،110xxxxx و...) را پیاده کنیم، و یک نسخه خیلی ساده‌تر با تنها یک بیت ادامه برای ما کافی است، بنابراین استفاده مستقیم از خود UTF-8 برای docIDها، نسبت به نیاز ما، over-engineering به حساب می‌آید.

پاسخ سوال ۳

1. با استفاده از تعریف gap (اولی نسبت به صفر، بقیه اختلاف با قبلی):
2. [56, 58, 63, 64, 67]⇒[56, 2, 5, 1, 3][56, 58, 63, 64, 67]⇒[56, 2, 5, 1, 3]
3. یعنی gap1 = 56، gap2 = 2، gap3 = 5، gap4 = 1، gap5 = 3.
4. الگوریتم کلی کدگذاری Variable-Byte با b=8b=8 برای عدد NN:
   - عدد NN را در مبنای ۲ می‌نویسیم.
   - رشته باینری را از راست به چپ به گروه‌های ۷ بیتی تقسیم می‌کنیم؛ اگر گروه چپ کمتر از ۷ بیت باشد، از چپ با صفر پُر می‌کنیم.
   - برای هر گروه به‌جز گروه آخر، یک بایت می‌سازیم که بیت اول آن ۰ (ادامه) و بقیه ۷ بیت همان گروه است.
   - برای آخرین گروه، بایت با بیت اول ۱ (پایان) و ۷ بیت بعدی همان گروه ساخته می‌شود.
   - پشت‌سرهم گذاشتن این بایت‌ها، نمایش Variable-Byte آن عدد است.
5. دیکودر دنباله بایت‌ها را از چپ به راست می‌خواند؛ برای هر عدد، تا وقتی بیت اول بایت‌ها ۰ است، ۷ بیت داده را به انتهای رشته جاری اضافه می‌کند، و وقتی به بایتی می‌رسد که بیت اول آن ۱ است، می‌فهمد نمایش این عدد تمام شده، کل بیت‌های جمع‌شده را به عدد ده‌دهی تبدیل می‌کند و به عنوان یک gap خروجی می‌دهد، سپس همین روند را برای عدد بعدی تکرار می‌کند؛ این کار نیاز به اشاره‌گر خارجی ندارد، چون بیت ادامه در خود بایت‌ها تمام اطلاعات مرزبندی را فراهم می‌کند.

پاسخ سوال ۴

1. وقتی تقریبا همه gapها بسیار کوچک باشند (مثلا اغلب در بازه ۱ تا ۷)، طرح A با $b=4$ بهتر فشرده می‌کند، چون با ۳ بیت داده می‌تواند این اعداد کوچک را در یک پکت کوتاه نمایش دهد، و سربار کلی (یک بیت کنترل + سه بیت داده) نسبت به استفاده از ۷ بیت داده در طرح B کمتر خواهد بود.
2. وقتی بسیاری از gapها بزرگ باشند (مثلا ده‌ها، صدها یا هزارها)، طرح B با $b=8$ معمولا سریع‌تر دیکود می‌شود، چون برای نمایش یک عدد بزرگ به پکت‌های کمتری نیاز دارد؛ هر عدد بزرگ در A باید به گروه‌های ۳بیتی شکسته شود که منجر به تعداد زیادی بایت/نیم‌بایت و کار بیشتر CPU در دیکود می‌شود.
3. طرحی مثل A بیشتر روی صرفه‌جویی حداکثری در بیت‌ها برای اعداد کوچک تمرکز می‌کند و در عوض کار CPU برای دیکود را افزایش می‌دهد؛ طرحی مثل B کمی از فشرده‌سازی بهینه فاصله می‌گیرد اما دیکود را ساده‌تر و سریع‌تر می‌کند. این دقیقا همان trade-off کلاسیک در فشرده‌سازی شاخص است: اگر I/O و فضای ذخیره‌سازی گلوگاه باشد می‌توان بیشتر فشرده کرد و CPU را درگیرتر کرد، ولی اگر CPU محدود باشد، معمولا طرح‌های ساده‌تر و کم‌هزینه‌تر مثل $b=8$ اولویت دارند.